JFP **32**, e8, 73 pages, 2022.

doi:10.1017/S095679682200003X

Denotational semantics as a foundation for cost recurrence extraction for functional languages

NORMAN DANNER
Wesleyan University, Middletown, CT 06459, USA
(*e-mail:* ndanner@wesleyan.edu)

DANIEL R. LICATA
Wesleyan University, Middletown, CT 06459, USA
(*e-mail:* dlicata@wesleyan.edu)

Abstract

A standard informal method for analyzing the asymptotic complexity of a program is to extract a recurrence that describes its cost in terms of the size of its input and then to compute a closed-form upper bound on that recurrence. We give a formal account of that method for functional programs in a higher order language with `let`-polymorphism. The method consists of two phases. In the first phase, a monadic translation is performed to extract a cost-annotated version of the original program. In the second phase, the extracted program is interpreted in a model. The key feature of this second phase is that different models describe different notions of size. This plays out in several ways. For example, when analyzing functions that take arguments of inductive types, different notions of size may be appropriate depending on the analysis. When analyzing polymorphic functions, our approach shows that one can formally describe the notion of size of an argument in terms of the data that is common to the notions of size for each type instance of the domain type. We give several examples of different models that formally justify various informal cost analyses to show the applicability of our approach.

1 Introduction

The method for analyzing the asymptotic cost of a (functional) program $f(x)$ that is typically taught to introductory undergraduate students is to extract a recurrence $T_f(n)$ that describes an upper bound on the cost of $f(x)$ in terms of the size of x and then establish a nonrecursive upper bound on $T_f(n)$ (we will focus on upper bounds, but much of what we say holds *mutatis mutandis* for lower bounds, and hence tight bounds). The goal of this work is to put the process of this informal approach to cost analysis on firm mathematical footing. Of course, various formalizations of cost analysis have been discussed for almost as long as there has been a distinct subfield of Programming Languages. Most of the recent work in this area is focused on developing formal techniques for cost analysis that enable the (possibly automated) analysis of as large a swath of programs as possible. In doing so, the type systems and the logics used grow ever more complex. There is work that incorporates size and cost into type information, for example, by employing refinement types or type-and-effect systems. There is work that formalizes reasoning

about cost in program logics such as separation logic with time credits. But as witnessed by most undergraduate texts on algorithm analysis, complex type systems and separation logic are not commonly taught. Instead, a function of some form (a *recurrence*) that computes the cost in terms of the size of the argument is extracted from the source code. This is the case for "simple" compositional worst-case analyses but also more more complex techniques. For example, the banker's and physicist's methods of amortized analysis likewise proceed by extracting a function to describe cost; the notion of cost itself, and the extraction of a suitably precise cost function, is more complex, but broadly speaking the structure of the analysis is the same. That is the space we are investigating here: how do we justify that informal process?[1] The justification might not itself play a role in applying the technique informally, any more than we require introductory students to understand the theory behind a type inference algorithm in order to informally understand why their programs typecheck. But certainly that theory should be settled. Our approach is through denotational semantics, which, in addition to justifying the informal process, also helps to explicate a few questions, such as why length is an appropriate measure of size for cost recurrences for polymorphic list functions (a question that is close to, but not quite the same as, parametricity).

Turning to the technical development, in previous work (Danner *et al*., 2013, 2015), we have developed a recurrence extraction technique for higher order functional programs for which the bounding is provable that is based on work by Danner & Royer (2007). The technique is described as follows:

1. We define what is essentially a monadic translation into the writer monad from a call-by-value *source language* that supports inductive types and structural recursion (fold) to a call-by-name *recurrence language*; we refer to programs in the latter language as *syntactic recurrences*. The recurrence language is axiomatized by a *size (pre)order* rather than equations. The syntactic recurrence extracted from a source-language program $f(x)$ describes both the cost and result of $f(x)$ in terms of x.
2. We define a *bounding relation* between source language programs and syntactic recurrences. The bounding relation is a logical relation that captures the notion that the syntactic recurrence is in fact a bound on the operational cost and the result of the source language program. This notion extends reasonably to higher type, where higher type arguments of a syntactic recurrence are thought of recurrences that are bounds on the corresponding arguments of the source language program. We then prove a *bounding theorem* that asserts that every typeable program in the source language is related to the recurrence extracted from it.
3. The syntactic recurrence is interpreted in a model of the recurrence language. This is where values are abstracted to some notion of size; e.g., the interpretation may be defined so that a value of inductive type δ is interpreted by the number of δ-constructors in v. We call the interpretation of a syntactic recurrence a *semantic recurrence*, and it is the semantic recurrences that are intended to match the recurrences that arise from informal analyses.

[1] We do mean the process here—some of the approaches do end up synthesizing recurrences, but that is almost a side effect rather than the first step.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

In this paper, we extend the above approach in several ways. First and foremost, we investigate the models, semantic recurrences, and size abstraction more thoroughly than in previous work and show how different models can be used to formally justify typical informal extract-and-solve cost analyses. Second, we add ML-style `let`-polymorphism and adapt the techniques to an environment-based operational semantics, a more realistic foundation for implementation than the substitution-based semantics used in previous work. In recent work, we have extended the technique for source languages with call-by-name and general recursion (Kavvos *et al*., 2020), and for amortized analyses (Cutler *et al*., 2020); we do not consider these extensions in the main body of this paper, in order to focus on the above issues in isolation.

Our source language, which we describe in Section 2, is a call-by-value higher order functional language with inductive datatypes and structural recursion (fold) and ML-style `let`-polymorphism. That is, `let`-bound identifiers may be assigned a quantified type, provided that type is instantiated at quantifier-free types in the body of the `let` expression. Restricting to polymorphism that is predicative (quantifiers can be instantiated only with nonquantified types), first-order (quantifiers range over types, not type constructors), and second-class (polymorphic functions cannot themselves be the input to other functions) is sufficient to program a number of example programs, without complicating the denotational models used to analyze them. We define an environment-based operational semantics where each rule is annotated with a cost. For simplicity, we only "charge" for each unfolding of a recursive call, but the technique extends easily for any notion of cost that can be defined in terms of evaluation rules. We could also replace the rule-based cost annotations with a "tick" construct that the programmer inserts at the code points for which a charge should be made, though this requires the programmer to justify the cost model.

The recurrence language in Section 3 is a call-by-name λ-calculus with explicit predicative polymorphism (via type abstraction and type application) and an additional type for costs. Ultimately, we care only about the meaning of a syntactic recurrence, not so much any particular strategy for evaluating it, and such a focus on mathematical reasoning makes call-by-name an appropriate formalism. The choice of explicit predicative polymorphism instead of `let`-polymorphism is minor, but arises from the same concerns: our main interest is in the models of the language, and it is simpler to describe models of the former than of the latter. To describe the recurrence language as call-by-name is not quite right because the verification of the bounding theorem that relates source programs to syntactic recurrences does not require an operational semantics. Instead it suffices to axiomatize the recurrence language by a preorder, which we call the size order. The size order is defined in Figure 11, and a brief glimpse will show the reader that the axioms primarily consist of directed versions of the standard call-by-name equations. This is the minimal set of axioms necessary to verify the bounding theorem, but as we discuss more fully when we investigate models of the recurrence language, there is more to the size order than that. In a nutshell, a model in which the size order axiom for a given type constructor is nontrivial (i.e., in which the two sides are not actually equal) is a model that genuinely abstracts that particular type constructor to a size.

We can think of the cost type in the recurrence language as the "output" of the writer monad, and the recurrence extraction function that we give in Section 4 as the call-by-value monadic translation of the source language. In some sense, then the recurrence extracted

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

from a source program is just a cost-annotated version of the program. However, we think of the "program" part of the syntactic recurrence differently: it represents the *size* of the source program. Thus, the syntactic recurrence simultaneously describes both the cost and size of the original program, what we refer to as a *complexity*. It is no surprise that we must extract both simultaneously, if for no other reason than compositionality, because if we are to describe cost in terms of size, then the cost of $f(g(x))$ depends on both the cost and size of $g(x)$. Thinking of recurrence extraction as a call-by-value monadic translation gives us insight into how to think of the size of a function: it is a mapping from sizes (of inputs) to complexities (of computing the result on an input of that size). This leads us to view size as a form of usage, or *potential* cost, and it is this last term that we adopt instead of size.[2] The bounding relation $e \preceq E$ that we define in Section 5 is a logical relation between source programs e and syntactic recurrences E. A syntactic recurrence is really a complexity, and $e \preceq E$ says that the operational cost of e is bounded by the cost component of E and that the value of e is bounded by the potential component of E. The Bounding Theorem (Theorem 1) tells us that every typeable program is bounded by the recurrence extracted from it. Its proof is somewhat long and technical, but follows the usual pattern for verifying the Fundamental Theorem for a logical relation, and the details are in Appendix 3.

In the recurrence language, the "data" necessary to describe the size has as much information as the original program; in the semantics we can abstract away as much or as little of this information as necessary. After defining environment models of the recurrence language in Section 6 (following Bruce *et al.*, 1990), in Section 7, we give several examples to demonstrate that different size abstractions result in semantic recurrences that formally justify typical extract-and-solve analyses. We stress that we are not attempting to analyze the cost of heretofore unanalyzed programs. Our goal is a formal process that mirrors as closely as possible the informal process we use at the board and on paper. The main examples demonstrate analyses where

1. The size of a value v of inductive type δ is defined in terms of the number of δ-constructors in v. For example, a list is measured by its length, a tree by either its size or its height, etc., enabling typical size-based recurrences (Section 7.2).
2. The size of a value v of inductive type δ is (more-or-less) the number of constructors of every inductive type in v. For example, the size of a `nat tree` t is the number of `nat tree` constructors in t (its usual "size") along with the maximum number of `nat` constructors in any node of t, enabling the analysis of functions with more complex costs, such as the function that sums the nodes of a `nat tree` (Section 7.3).
3. A polymorphic function can be analyzed in terms of a notion of size that is more abstract than that given by its instances (Section 7.4). For example, while the size of a `nat tree` may be a pair (k, n), where k is the maximum key value and n the size (e.g., to permit analysis of the function that sums all the nodes), we may want the domain of the recurrence extracted from a function of type α `tree` $\to \rho$ to be $\mathbf{N}$, corresponding to counting only the `tree` constructors.

[2] We warn the reader that "potential" as we use it here is not related to "potential" as it is used in amortized analysis, though it does seem like potential associated to a data value gives information about its use cost, so there may be a deeper connection; we leave this question for future study.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

4. We make use of the fact that the interpretation of the size order just has to satisfy certain axioms to derive recurrences for *lower bounds*. As an example, we parlay this into a formal justification for the informal argument that $\texttt{map}(f \circ g)$ is more efficient than $(\texttt{map}\, f) \circ (\texttt{map}\, g)$ (Section 7.5).

These examples end up clarifying the role of the size order, as mentioned earlier. It is not just the rules necessary to drive the proof of the syntactic bounding theorem, but a nontrivial interpretation of $\leq_\sigma$ (i.e., one in which $e \leq_\sigma e'$ is valid but not $e = e'$) tells us that we have a model with a nontrivial size abstraction for σ. This clarification highlights interesting analogies with abstract interpretation: (1) when a datatype $\delta = \mu t.F$ is interpreted by a nontrivial size abstraction, there is an abstract interpretation between $[\![\delta]\!]$ (abstract) and $[\![F[\delta]]\!]$ (concrete) and (2) interpreting the recurrence extracted from a polymorphic function in terms of a more abstract notion of size is possible if there is an abstract interpretation between two models.

The remaining sections of the paper discuss recent work in cost analysis and how our work relates to it, as well as limitations of and future directions for our approach.

2 The source language

The source language that serves as the object language of our recurrence extraction technique is a higher order language with inductive types, structural iteration (fold) over those types, and ML-style polymorphism (i.e., predicative polymorphism, with polymorphic identifiers introduced only in let bindings), with an environment-based operational semantics that approximates typical implementation. This generalizes the source language of Danner *et al*. (2015), which introduced the technique for a monomorphically typed language with a substitution-based semantics. We address general recursion in Section 8.

The grammar and typing rules for expressions are given in Figure 1. Type assignment derives (quantifier-free) types for expressions given a *type context* that assigns type schemes (quantified types) to identifiers. We write _ for the empty type context. Values are not a subset of expressions because, as one would expect in implementation, a function value consists of a function expression along with a *value environment*: a binding of free variables to values. The same holds for values of suspension type, and we refer to any pair of an expression and a value environment for its free variables as a *closure* (thus we use closure more freely than the usual parlance, in which it is restricted to functions). We adopt the notation common in the explicit substitution literature (e.g., Abadi *et al*., 1991) and write $v\theta$ for a closure with value environment θ. Since the typing for map and mapv expressions depend on values, this requires a separate notion of typing for values, which in turn depends on a notion of typing for closures. These are defined in Figures 2 and 3. There is nothing deep in the typing of a closure value $v\theta$ under context Γ. Morally, the rules just formalize that v can be assigned the expected type without regard to θ and that $\theta(x)$ is of type $\Gamma(x)$. But since type contexts may assign type schemes, whereas type assignment only derives types, the formal definition is that $\theta(x)$ can be assigned any instance of $\Gamma(x)$.

We will freely assume notation for n-ary sums and products and their corresponding introduction and elimination forms, such as $\sigma_0 \times \sigma_1 \times \sigma_2$, (e_0, e_1, e_2), and $\pi_1\, e$ for

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$
\begin{array}{rcll}
\rho, \sigma \in \text{Type} & ::= & \alpha \mid \texttt{unit} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \to \sigma \mid \sigma\ \texttt{susp} \mid \delta & \text{(types)} \\
\delta & ::= & \mu t.F & \text{(inductive types)} \\
F & ::= & t \mid \alpha \mid \sigma \mid F \times F \mid F + F \mid \sigma \to F & \text{(shape functors)} \\
\tau & ::= & \sigma \mid \forall \alpha.\tau & \text{(type schemes)} \\
e & ::= & x \mid ( ) \mid (e, e) \mid \pi_i\, e \mid \iota_i\, e \mid \texttt{case}\ e\ \texttt{of}\ x.e_0; x.e_1 & \\
& \mid & \lambda x.e \mid e\ e \mid \texttt{delay}\ e \mid \texttt{force}\ e & \\
& \mid & \texttt{c}_\delta\ e \mid \texttt{d}_\delta\ e \mid \texttt{fold}_\delta\ e\ \texttt{of}\ x.e & \\
& \mid & \texttt{let}\ x = e\ \texttt{in}\ e & \\
& \mid & \texttt{map}_F\ y.v\ \texttt{into}\ e \mid \texttt{mapv}_F\ y.v\ \texttt{into}\ v & \text{(expressions)}
\end{array}
$$

$$\frac{}{\Gamma, x : \forall \vec{\alpha}.\sigma \vdash x : \sigma\{\vec{\alpha} \mapsto \vec{\sigma}\}} \qquad \frac{}{\Gamma \vdash ( ) : \texttt{unit}}$$

$$\frac{\Gamma \vdash e_0 : \sigma_0 \qquad \Gamma \vdash e_1 : \sigma_1}{\Gamma \vdash (e_0, e_1) : \sigma_0 \times \sigma_1} \qquad \frac{\Gamma \vdash e : \sigma_0 \times \sigma_1}{\Gamma \vdash \pi_i\, e : \sigma_i}$$

$$\frac{\Gamma \vdash e : \sigma_i}{\Gamma \vdash \iota_i\, e : \sigma_0 + \sigma_1} \qquad \frac{\Gamma \vdash e : \sigma_0 + \sigma_1 \qquad \{\Gamma, x : \sigma_i \vdash e_i : \sigma_i\}_{i=0,1}}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ x.e_0; x.e_1 : \sigma}$$

$$\frac{\Gamma, x : \sigma' \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \sigma' \to \sigma} \qquad \frac{\Gamma \vdash e : \sigma' \to \sigma \qquad \Gamma \vdash e' : \sigma'}{\Gamma \vdash e\ e' : \sigma}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \texttt{delay}\ e : \sigma\ \texttt{susp}} \qquad \frac{\Gamma \vdash e : \sigma\ \texttt{susp}}{\Gamma \vdash \texttt{force}\ e : \sigma}$$

$$\frac{\Gamma \vdash e : F[\delta]}{\Gamma \vdash \texttt{c}_\delta\ e : \delta}\ (\delta = \mu t.F) \qquad \frac{\Gamma \vdash e : \delta}{\Gamma \vdash \texttt{d}_\delta\ e : F[\delta]}\ (\delta = \mu t.F)$$

$$\frac{\Gamma \vdash e' : \delta \qquad \Gamma, x : F[\sigma\ \texttt{susp}] \vdash e : \sigma}{\Gamma \vdash \texttt{fold}_\delta\ e'\ \texttt{of}\ x.e : \sigma}\ (\delta = \mu t.F)$$

$$\frac{\Gamma \vdash e' : \sigma' \qquad \Gamma, x : \forall \vec{\alpha}.\sigma' \vdash e : \sigma \qquad \vec{\alpha}\ \text{not free in any}\ \Gamma(y)}{\Gamma \vdash \texttt{let}\ x = e'\ \texttt{in}\ e : \sigma}$$

$$\frac{y : \rho \vdash v : \sigma \qquad \Gamma \vdash e : F[\rho]}{\Gamma \vdash \texttt{map}_F\ y.v\ \texttt{into}\ e : F[\sigma]} \qquad \frac{y : \rho \vdash v' : \sigma \qquad \_ \vdash v : F[\rho]}{\_ \vdash \texttt{mapv}_F\ y.v'\ \texttt{into}\ v : F[\sigma]}$$

Fig. 1: A source language with let polymorphism and inductive datatypes. `map` and `mapv` expressions depend on values, which are defined in Figure 2.

$(\sigma_0 \times \sigma_1) \times \sigma_2$, $((e_0, e_1), e_2)$, and $\pi_1\,(\pi_0\, e)$, respectively. We write $\mathrm{fv}(e)$ for the free variables of e and $\mathrm{ftv}(\tau)$ for the free type variables of τ.

Inductive types are defined by shape functors, ranged over by the metavariable F; a generic inductive type has the form $\mu t.F$. If F is a shape functor and σ a type, then $F[\sigma]$ is the result of substituting σ for free occurrences of t in F (the μ operator binds t, of course). Formally a shape functor is just a type, and so when certain concepts are defined by induction on type, they are automatically defined for shape functors as well. In the syntax for shape functors, t is a fixed type variable, and hence simultaneous nested definitions are not allowed. That is, types such as $\mu t.\mathrm{unit} + \mu s.(\mathrm{unit} + s \times t)$ are forbidden. However, an

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\begin{array}{lcl} v & ::= & y \mid ( ) \mid (v, v) \mid \iota_i\, v \mid (\lambda x.e)\theta \mid (\texttt{delay}\, e)\theta \mid \mathrm{c}_\delta\,(v) \\ \theta & ::= & \mathrm{TmVar} \to_{\mathrm{fin}} \mathrm{Val} \end{array}$$

$$\frac{}{\Gamma, y : \sigma \vdash y : \sigma} \qquad \frac{}{\Gamma \vdash ( ) : \texttt{unit}}$$

$$\frac{\{\Gamma \vdash v_i : \sigma_i\}_{i=0,1}}{\Gamma \vdash (v_0, v_1) : \sigma_0 \times \sigma_1} \qquad \frac{\Gamma \vdash v : \sigma_i}{\Gamma \vdash \iota_i\,(v) : \sigma_0 + \sigma_1}$$

$$\frac{\Gamma \vdash (\lambda x.e)\theta : (\sigma' \to \sigma)\ \texttt{closure}}{\Gamma \vdash (\lambda x.e)\theta : \sigma' \to \sigma} \qquad \frac{\Gamma \vdash (\texttt{delay}\, e)\theta : (\sigma\ \texttt{susp})\ \texttt{closure}}{\Gamma \vdash (\texttt{delay}\, e)\theta : \sigma\ \texttt{susp}}$$

$$\frac{\Gamma \vdash v : F[\delta]}{\Gamma \vdash \mathrm{c}_\delta\, v : \delta}\ (\delta = \mu t.F)$$

Fig. 2: Grammar and typing rules for values. For any value environment θ, it must be that for all x there is σ such that $\_ \vdash \theta(x) : \sigma$. The judgment $\Gamma \vdash e\theta : \sigma\ \texttt{closure}$ is defined in Figure 3.

$$\frac{\text{For all } x \in \mathrm{dom}\,\theta\text{: if } \Gamma(x) = \forall \vec{\alpha}.\rho \text{ then } \forall \vec{\sigma}, \_ \vdash \theta(x) : \rho\{\vec{\sigma}/\vec{\alpha}\}}{\theta \text{ is a } \Gamma\text{-environment}}$$

$$\frac{\Gamma, \Gamma' \vdash e : \sigma \qquad \theta \text{ is a } \Gamma'\text{-environment}}{\Gamma \vdash e\theta : \sigma\ \texttt{closure}}$$

Fig. 3: Typing for closures.

$$\begin{array}{rcl} \texttt{bool} & = & \texttt{unit} + \texttt{unit} \\ \texttt{false} & = & \iota_0^{\texttt{bool}}\,( ) \\ \texttt{true} & = & \iota_1^{\texttt{bool}}\,( ) \end{array} \qquad \begin{array}{rcl} \texttt{nat} & = & \mu t.\texttt{unit} + t \\ \mathrm{Z} & = & \mathrm{c}_{\texttt{nat}}\,(\iota_0\,( )) \\ \mathrm{S}\,x & = & \mathrm{c}_{\texttt{nat}}\,(\iota_1\,x) \\ \underline{n} & = & \mathrm{S}(\ldots\mathrm{SZ})\ (n\ \mathrm{S's}) \end{array}$$

$$\begin{array}{rcl} \sigma\ \texttt{list} & = & \mu t.\texttt{unit} + \sigma \times t \\ \texttt{nil}_\sigma & = & \mathrm{c}_{\sigma\ \texttt{list}}\,(\iota_0\,( )) \\ \texttt{cons}_\sigma(x, xs) & = & \mathrm{c}_{\sigma\ \texttt{list}}\,(\iota_1\,((x, xs))) \end{array} \qquad \begin{array}{rcl} \sigma\ \texttt{tree} & = & \mu t.\texttt{unit} + \sigma \times t \times t \\ \texttt{emp}_\sigma & = & \mathrm{c}_{\sigma\ \texttt{tree}}\,(\iota_0\,( )) \\ \texttt{node}_\sigma(x, t_0, t_1) & = & \mathrm{c}_{\sigma\ \texttt{tree}}\,(\iota_1\,(x, t_0, t_1)) \end{array}$$

Fig. 4: Some standard types in the source language.

inductive type can be used inside of other types via the constant functor (σ) production of F, e.g. coding the type $(\alpha\ \texttt{list})\ \texttt{list}$ as $\mu t.\mathrm{unit} + (\mu t.\mathrm{unit} + \alpha \times t) \times t$. This restriction is just to simplify the presentation of the languages and models, and lifting it does not require fundamental changes. Figure 4 gives a number of types and values that we will use in examples. We warn the reader that because most models of the recurrence language have nonstandard interpretations of inductive types, the types σ and $\mu t.\sigma$ may be treated very differently even when t is not free in σ. Thus, it can actually make a real difference whether we define $\texttt{bool}$ to be $\texttt{unit} + \texttt{unit}$ or $\mu t.\texttt{unit} + \texttt{unit}$ (if every type that would be defined by an ML `datatype` declaration were implemented as a possibly degenerate inductive type).

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

For every inductive type $\delta = \mu t.F$ there is an associated constructor $\mathtt{c}_\delta$, destructor $\mathtt{d}_\delta$, and iterator $\mathtt{fold}_\delta$. Thought of informally as term constants, the first two have the typical types $F[\delta] \to \delta$ and $\delta \to F[\delta]$, but the type of $\mathtt{fold}_\delta$ is somewhat nonstandard: $\delta \to (F[\sigma\ \mathtt{susp}] \to \sigma) \to \sigma$. We use suspension types of the form $\sigma\ \mathtt{susp}$ primarily to delay computation of recursive calls in evaluating `fold` expressions. This is not necessary for any theoretical concerns, but rather practical: without something like this, implementations of standard programs would have unexpected costs. We will return to this when we discuss the operational semantics. We also observe that in this informal treatment, the types are not polymorphic; in our setting, polymorphism and inductive types are orthogonal concerns.

The $\mathtt{map}_F$ and $\mathtt{mapv}_F$ constructors are used to define the operational semantics of $\mathtt{fold}_{\mu t.F}$. The latter witnesses functoriality of shape functors. Informally speaking, evaluation of $\mathtt{mapv}_F\, y.v'\ \mathtt{into}\ v$ traverses a value v of type $F[\delta]$, applying a function $y \mapsto v'$ of type $\delta \to \sigma$ to each inductive subvalue of type δ to obtain a value of type $F[\sigma]$. $\mathtt{map}_F$ is a technical tool for defining this action when F is an arrow shape, in which case the value of type $F[\delta]$ is really a delayed computation and hence is represented by an arbitrary expression. Because the definition of $\mathtt{map}_F$ and $\mathtt{mapv}_F$ depend on values, we must define them (and their typing) simultaneously with terms. Furthermore, evaluation of $\mathtt{mapv}_{\rho \to F}$ results in a function closure value that contains a $\mathtt{map}_F$ expression, and the function closure itself is, as usual, an ordinary λ-expression. This is also the reason that `map` and `mapv` are part of the language, rather than just part of the metalanguage used to define the operational semantics. They are not intended to be used in program definitions though, so we make the following definition:

Definition (Core language)**.** *The* core language *consists of the terms of the source language that are typeable not using* `map` *or* `mapv`.

The operational cost semantics for the language is defined in Figure 5 and its dependencies, which define a relation $e\theta \downarrow^n v$, where e is a (well-typed) expression, θ a value environment, v a value, and n a non-negative integer. As with closure values, we write a closure with expression e and value environment θ as $e\theta$, and opt for this notation for compactness (a more typically presentation might be $\theta \vdash e \downarrow^n v$). The intended meaning is that under value environment θ, the term e evaluates to the value v with cost n. A value environment that needs to be spelled out will be written $\{x_0 \mapsto v_0, \ldots, x_{n-1} \mapsto v_{n-1}\}$ or more commonly $\{\vec{x} \mapsto \vec{v}\}$. We write $\theta\{y \mapsto v\}$ for extending a value environment θ by binding the (possibly fresh) variable y to v. Value environments are part of the language, so when we write $e\theta$, the bindings are not immediately applied. However, we use a substitution notation $\{v/y\}$ for defining the semantics of $\mathtt{mapv}_t$ because this is defined in the metalanguage as a metaoperation. Using explicit environments, rather than a metalanguage notation for substitutions, adds a certain amount of syntactic complexity. The payoff is a semantics that more closely reflects typical implementation.

Our approach to charging some amount of cost for each step of the evaluation, where that amount may depend on the main term former, is standard. Recurrence extraction is parametric in these choices. We observe that our environment-based semantics permits

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\frac{}{x\theta \downarrow^0 \theta(x)} \qquad \frac{}{()\theta \downarrow^0 ()}$$

$$\frac{e_0\theta \downarrow^{n_0} v_0 \qquad e_1\theta \downarrow^{n_1} v_1}{(e_0, e_1)\theta \downarrow^{n_0+n_1} (v_0, v_1)} \qquad \frac{e\theta \downarrow^n (v_0, v_1)}{(\pi_i\, e)\theta \downarrow^n v_i}$$

$$\frac{e\theta \downarrow^n v}{(\iota_i\, e)\theta \downarrow^n \iota_i\, v} \qquad \frac{e\theta \downarrow^n \iota_i\, v_i \qquad e_i\theta\{x \mapsto v_i\} \downarrow^{n_i} v}{(\mathtt{case}\ e\ \mathtt{of}\ x.e_0; x.e_1)\theta \downarrow^{n+n_i} v}$$

$$\frac{}{(\lambda x.e)\theta \downarrow^0 (\lambda x.e)\theta} \qquad \frac{e_0\theta \downarrow^{n_0} (\lambda x.e_0')\theta' \qquad e_1\theta \downarrow^{n_1} v_1 \qquad e_0'\theta'\{x \mapsto v_1\} \downarrow^n v}{(e_0\ e_1)\theta \downarrow^{n_0+n_1+n} v}$$

$$\frac{}{(\mathtt{delay}\ e)\theta \downarrow^0 (\mathtt{delay}\ e)\theta} \qquad \frac{e\theta \downarrow^n (\mathtt{delay}\ e')\theta' \qquad e'\theta' \downarrow^{n'} v}{(\mathtt{force}\ e)\theta \downarrow^{n+n'} v}$$

$$\frac{e\theta \downarrow^n v}{(\mathtt{c}_\delta\ e)\theta \downarrow^n \mathtt{c}_\delta\ v} \qquad \frac{e\theta \downarrow^n \mathtt{c}_\delta\ v}{(\mathtt{d}_\delta\ e)\theta \downarrow^n v}$$

$$\frac{e'\theta \downarrow^{n'} \mathtt{c}_\delta\ v' \qquad \mathtt{mapv}_F\ y.(\mathtt{delay}\ (\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e))\theta\ \mathtt{into}\ v' \downarrow v'' \qquad e\theta\{x \mapsto v''\} \downarrow^n v}{(\mathtt{fold}_\delta\ e'\ \mathtt{of}\ x.e)\theta \downarrow^{n'+n+1} v}$$

$$\frac{e'\theta \downarrow^{n'} v' \qquad e\theta\{x \mapsto v'\} \downarrow^n v}{(\mathtt{let}\ x = e'\ \mathtt{in}\ e)\theta \downarrow^{n'+n} v}$$

Fig. 5: The operational cost semantics for the source language. We only define evaluation for closures $e\theta$ such that $\_ \vdash e\theta : \sigma$ for some σ, and hence just write $e\theta$. The semantics for `fold` depends on the semantics for `map`, which is given in Figure 6.

$$\frac{e\theta \downarrow^n v'' \qquad \mathtt{mapv}_F\ y.v'\ \mathtt{into}\ v'' \downarrow v}{(\mathtt{map}_F\ y.v'\ \mathtt{into}\ e)\theta \downarrow^n v}$$

$$\frac{}{\mathtt{mapv}_t\ y.v'\ \mathtt{into}\ v \downarrow v'\{v/y\}} \qquad \frac{}{\mathtt{mapv}_\sigma\ y.v'\ \mathtt{into}\ v \downarrow v}$$

$$\frac{\{\mathtt{mapv}_{F_i}\ y.v'\ \mathtt{into}\ v_i' \downarrow v_i\}_{i=0,1}}{\mathtt{mapv}_{F_0 \times F_1}\ y.v'\ \mathtt{into}\ (v_0', v_1') \downarrow (v_0, v_1)} \qquad \frac{\mathtt{mapv}_{F_i}\ y.v'\ \mathtt{into}\ v \downarrow v_i}{\mathtt{mapv}_{F_0+F_1}\ y.v'\ \mathtt{into}\ \iota_i\, v \downarrow \iota_i\, v_i}$$

$$\frac{}{\mathtt{mapv}_{\rho \to F}\ y.v'\ \mathtt{into}\ (\lambda x.e)\theta \downarrow (\lambda x.\mathtt{map}_F\ y.v'\ \mathtt{into}\ e)\theta}$$

Fig. 6: The operational semantics for the source language `map` and `mapv` constructors. Substitution of values is defined in Figure 7.

$$y\{v/y\} = v \qquad ((\lambda x.e)\theta)\{v/y\} = (\lambda x.e)(\theta\{y \mapsto v\})$$

$$()\{v/y\} = () \qquad ((\mathtt{delay}\ e)\theta)\{v/y\} = (\mathtt{delay}\ e)(\theta\{y \mapsto v\})$$

$$(v_0, v_1)\{v/y\} = (v_0\{v/y\}, v_1\{v/y\}) \qquad (\mathtt{c}_\delta\ v')\{v/y\} = \mathtt{c}_\delta\ (v'\{v/y\})$$

$$(\iota_i\, v')\{v/y\} = \iota_i\, (v'\{v/y\})$$

Fig. 7: Substitution of values for identifiers in values, $v'\{v/y\}$.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

```
fun member(t : bst, x : int) : bool =
    case t of
         E => false
       | N(y, t0, t1) => if x = y then true
                         else if x < y then member(t0, x)
                         else member(t1, x)
```

(a) Binary search tree membership function in ML.

```
fun member(t : bst, x : int) : bool =
    case t of
         E => false
       | N(y, t0, t1) => if x = y then true
                         else if x < y then member(t0, x)
                         else member(t1, x)
```

(b) Binary search tree membership without suspensions.

```
fun member(t, x) =
    treefold (fn (y, r0, r1) => if x = y then true
                                else if x < y then force r0
                                else force r1) t
```

(c) Binary search tree membership with suspensions.

Fig. 8: Using suspension types to control evaluation of recursive calls in fold-like constructs.

us to charge even for looking up the value of an identifier, something that is difficult to codify in a substitution-based semantics. Our particular choice to charge one unit of cost for each unfolding of a `fold`, and no cost for any other form, is admittedly ad hoc, but gives expected costs with a minimum of bookkeeping fuss, especially when it comes to the semantic interpretations of the recurrences. Another common alternative is to define a tick operation $\texttt{tick} : \alpha \to \alpha$ which charges a unit of cost, as done by Danielsson (2008) and others. This requires the user to annotate the code at the points for which cost should be charged, which increases the load on the programmer, but allows her to be specific about exactly what to count (e.g., only comparisons). It is straightforward to adapt our approach to that setting.

The reason for suspending the recursive call in the semantics of `fold` is to ensure that typical recursively defined functions that do not always evaluate all recursive calls still have the expected cost. For example, consider membership testing in a binary search tree. Typical ML code for such a function might look something like the code in Figure 8(a). This function is linear in the height of t because the lazy evaluation of conditionals ensures that at most one recursive call is evaluated. If we were to implement—member—with a—fold—operator for trees that does not suspend the recursive call (so the step function would have type $\texttt{int} \times \texttt{bool} \times \texttt{bool} \to \texttt{bool}$), as in Figure 8(b), then the recursive calls—r0—and—r1—are evaluated at each step, leading to a cost that is linear in the size of t, rather than the height. Our solution is to ensure that the recursive calls are delayed, and only evaluated when the corresponding branch of the conditional is evaluated, so the step function

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

has type $\texttt{int} \times \texttt{bool susp} \times \texttt{bool susp} \to \texttt{bool}$, in which case the code looks something like that of Figure 8(c). This issue does not come up when recursive definitions are allowed only at function type, as is typical in call-by-value languages. However, in order to simplify the construction of models, here we restrict recursive definitions to the use of $\texttt{fold}_\delta$ (i.e., to structural recursion only, rather than general recursion), which must be permitted to have any result type.

Given the complexity of the language, it behooves us to verify type preservation. For this cost is irrelevant, so we write $e\theta \downarrow v$ to mean $e\theta \downarrow^n v$ for some n. Remember that we our notion of closure includes expressions of any type, so this theorem does not just state type preservation for functions.

Theorem 1 (Type preservation)**.** *If* $\_ \vdash e\theta : \sigma$ *and* $e\theta \downarrow v$*, then* $\vdash v : \sigma$.

Proof See Appendix 1. ■

3 The recurrence language

The recurrence language is defined in Figure 9. It is a standard system of predicative polymorphism with explicit type abstraction and application. Most of the time we will elide type annotations from variable bindings, mentioning them only when demanded for clarity. The types and terms corresponding to those of Figure 4 are given in Figure 10. This is the language into which we will extract syntactic recurrences from the source language. A syntactic recurrence is more-or-less a cost-annotated version of a source language program. As we are interested in the value (denotational semantics) of the recurrences and not in operational considerations, we think of the recurrence language in a more call-by-name way (although, as we will see, the main mode of reasoning is with respect to an ordering, rather than equality).

3.1 The cost type

The recurrence language has a *cost type* C. As we discuss in Section 4, we can think of recurrence extraction as a monadic translation into the writer monad, where the "writing" action is to increment the cost component. Thus it suffices to ensure that C is a monoid, though in our examples of models, it is usually interpreted as a set with more structure (e.g., the natural numbers adjoined with an "infinite" element).

3.2 The "missing" pieces from the source language

There are no suspension types, nor term constructors corresponding to `let`, `map`, or `mapv`. We are primarily interested in the denotations of expressions in the recurrence language, not in carefully accounting for the cost of evaluating them. Of course, it is convenient to have the standard syntactic sugar $\mathsf{let}\, x_0 = e_0, \ldots, x_{n-1} = e_{n-1}\, \mathsf{in}\, e$ for

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$
\begin{array}{rcll}
\rho, \sigma & ::= & \alpha \mid \mathsf{C} \mid \mathsf{unit} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \to \sigma \mid \delta & \text{(types)} \\
\delta & ::= & \mu t.F & \text{(inductive types)} \\
F & ::= & t \mid \sigma \mid F \times F \mid F + F \mid \sigma \to F & \text{(shape functors)} \\
\tau & ::= & \sigma \mid \forall \alpha.\tau & \text{(type schemes)} \\
e & ::= & x \mid 0 \mid 1 \mid e + e \mid (\,) \mid (e, e) \mid \pi_i\, e & \\
 & & \mid \iota_i\, e \mid \mathsf{case}\ e\ \mathsf{of}\ (x : \sigma_0).e_0; (x : \sigma_1).e_1 & \\
 & \mid & \lambda(x : \sigma).e \mid e\ e \mid \Lambda\alpha.e \mid e\ \sigma & \\
 & \mid & \mathsf{c}_\delta \mid \mathsf{d}_\delta \mid \mathsf{fold}_\delta & \text{(expressions)}
\end{array}
$$

$$\frac{}{\Gamma \vdash 0 : \mathsf{C}} \qquad \frac{}{\Gamma \vdash 1 : \mathsf{C}} \qquad \frac{\Gamma \vdash e_0 : \mathsf{C} \qquad \Gamma \vdash e_1 : \mathsf{C}}{\Gamma \vdash e_0 + e_1 : \mathsf{C}}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash (\,) : \mathsf{unit}}$$

$$\frac{\Gamma \vdash e_0 : \sigma_0 \qquad \Gamma \vdash e_1 : \sigma_1}{\Gamma \vdash (e_0, e_1) : \sigma_0 \times \sigma_1} \qquad \frac{\Gamma \vdash e : \sigma_0 \times \sigma_1}{\Gamma \vdash \pi_i\, e : \sigma_i}$$

$$\frac{\Gamma \vdash e : \sigma_i}{\Gamma \vdash \iota_i\, e : \sigma_0 + \sigma_1} \qquad \frac{\Gamma \vdash e : \sigma_0 + \sigma_1 \qquad \{\Gamma, x : \sigma_i \vdash e_i : \sigma_i\}_{i=0,1}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{(x : \sigma_i).e_i\}_{i=0,1} : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash e : \sigma'}{\Gamma \vdash \lambda(x : \sigma).e : \sigma \to \sigma'} \qquad \frac{\Gamma \vdash e : \sigma' \to \sigma \qquad \Gamma \vdash e' : \sigma'}{\Gamma \vdash e\ e' : \sigma}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \alpha \notin \mathrm{ftv}(\Gamma) \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e\ \sigma : \tau\{\sigma/\alpha\}}\ (\sigma \text{ quantifier-free})$$

$$\frac{\Gamma \vdash e : F[\delta]}{\Gamma \vdash \mathsf{c}_\delta\, e : \delta}\ (\delta = \mu t.F) \qquad \frac{\Gamma \vdash e : \delta}{\Gamma \vdash \mathsf{d}_\delta\, e : F[\delta]}\ (\delta = \mu t.F)$$

$$\frac{\Gamma \vdash e' : \delta \qquad \Gamma, x : F[\sigma] \vdash e : \sigma}{\Gamma \vdash \mathsf{fold}_\delta\, e'\ \mathsf{of}\ (x : F[\sigma]).e : \sigma}\ (\delta = \mu t.F)$$

Fig. 9: The recurrence language grammar and typing.

$$
\begin{array}{rr}
\mathsf{bool} = \mathsf{unit} + \mathsf{unit} & \mathsf{nat} = \mu t.\mathsf{unit} + t \\
\mathsf{false} = \iota_0^{\mathsf{bool}}\,(\,) & \mathsf{Z} = \mathsf{c}_{\mathsf{nat}}\,(\iota_0\,(\,)) \\
\mathsf{true} = \iota_1^{\mathsf{bool}}\,(\,) & \mathsf{S}\,x = \mathsf{c}_{\mathsf{nat}}\,(\iota_1\,x)
\end{array}
$$

$$
\begin{array}{rr}
\sigma\ \mathsf{list} = \mu t.\mathsf{unit} + \sigma \times t & \sigma\ \mathsf{tree} = \mu t.\mathsf{unit} + \sigma \times t \times t \\
\mathsf{nil}_\sigma = \Lambda\vec{\alpha}.\mathsf{c}_{\sigma\ \mathsf{list}}\ \vec{\alpha}(\iota_0\,(\,)) & \mathsf{emp}_\sigma = \Lambda\vec{\alpha}.\mathsf{c}_{\sigma\ \mathsf{tree}}\ \vec{\alpha}(\iota_0\,(\,)) \\
\mathsf{cons}_\sigma(x, xs) = \Lambda\vec{\alpha}.\mathsf{c}_{\sigma\ \mathsf{list}}\ \vec{\alpha}(\iota_1\,((x, xs))) & \mathsf{node}_\sigma(x, t_0, t_1) = \Lambda\vec{\alpha}.\mathsf{c}_{\sigma\ \mathsf{tree}}\ \vec{\alpha}(\iota_1\,(x, t_0, t_1))
\end{array}
$$

Fig. 10: Some standard types in the recurrence language corresponding to those in Figure 4.

$e\{e_0, \ldots, e_{n-1}/x_0, \ldots, x_{n-1}\}$. Because of the way in which the size order is axiomatized, this must be defined as a substitution, not as a β-expansion. Likewise, we still need a construct that witnesses functoriality of shape functors, but it suffices to do so with a metalanguage macro $F[(x : \rho).e', e]$ that is defined in Figure 12.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

3.3 Datatype constructor, destructor, and fold

The constructor, destructor, and fold terms are similar to those in the source language, though here we use the more typical type for fold_δ. Since type abstraction and application are explicit in the recurrence language, it may feel a bit awkward that β-reduction for types seems to change these constants; for example,$(\Lambda\alpha.\cdots \mathsf{fold}_{\alpha\ \mathsf{list}}\ e'\ \mathsf{of}\ x.e\cdots)\,\sigma$ would convert to $\cdots \mathsf{fold}_{\sigma\ \mathsf{list}}\ e'\ \mathsf{of}\ x.e\cdots$. The right way to think of this is that there is really a single constant fold that maps inductive types to the corresponding operator—in effect, we write fold_δ for fold δ, and so the substitution of a type for a type variable does not change the constant, but rather the argument to fold.

The choice as to whether to implement datatype-related constructs as term formers or as constants and whether they should be polymorphic or not is mostly a matter of convenience. The choice here meshes better with the definitions of environment models we use in Section 6, but using constants does little other than force us to insert some semantic functions into some definitions. However, one place where this is not quite the whole story is for fold_δ, which one might be tempted to implement as a term constant of type $\forall\vec{\alpha}\beta.(F[\beta]\to\beta)\to\delta\to\beta$, where $\vec{\alpha}=\mathrm{ftv}(\delta)$. The typing we have chosen is equivalent with respect to any standard operational or denotational semantics. However, our denotational semantics will be non-standard, and the choice turns out to matter in the model construction of Section 7.4. There, we show how to identify type abstraction with size abstraction in a precise sense. Were we to use the polymorphic type for fold_δ, then even when δ is monomorphic, fold_δ would still have polymorphic type (for the result), and that would force us to perform undesirable size abstraction on values of the monomorphic type.

3.4 The size order

The semantics of the recurrence language is described in terms of size orderings $\leq_\tau$ that is defined in Figure 11 for each type τ (although the rules only define a preorder, we will continue to refer to it as an order). The syntactic recurrence extracted from a program of type σ has the type $\mathsf{C}\times\langle\!\langle\sigma\rangle\!\rangle$. The intended interpretation of $\langle\!\langle\sigma\rangle\!\rangle$ is the set of sizes of source language values of type σ. We expect to be able to compare sizes, and that is the role of $\leq_\sigma$. Although $\leq_{\mathsf{C}}$ is more appropriately thought of as an ordering on costs, general comments about $\leq$ apply equally to $\leq_{\mathsf{C}}$, so we describe it as a size ordering as well to reduce verbosity.[3] The relation $\leq_{\mathsf{C}}$ just requires that C be a monoid (i.e., have an associative operation with an identity). In particular, there are no axioms governing 1 needed to prove the bounding theorem; it is not even necessary to require that $0\neq 1$ or even $0\leq_{\mathsf{C}} 1$.

Let us gain some intuition behind the axioms for $\leq_\sigma$. On the one hand, they are just directed versions of the standard call-by-name equational calculus that one might expect. In the proof of the Syntactic Bounding Theorem (Theorem 5), that is the role they play. But there is more going on here than that. The intended interpretation of the axioms is that the introduction-elimination round-trips that they describe provide a possibly less precise

[3] Avanzini & Dal Lago (2017) perform cost analysis by representing execution cost directly and then measuring the size of the result, so thinking of C as a set of sizes (of costs?) may not be unreasonable.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\mathscr{C} ::= [\,] \mid \mathscr{C} + e \mid e + \mathscr{C} \mid \pi_i\,\mathscr{C} \mid \mathsf{case}\,\mathscr{C}\,\mathsf{of}\,\{(x:\sigma_i).e_i\}_{i=0,1} \mid \mathscr{C}\;e \mid \mathsf{d}_\delta\,\mathscr{C} \mid \mathsf{fold}_\delta\,\mathscr{C}\,\mathsf{of}\,x.e$$

$$\frac{}{\Gamma \vdash e \leq_\tau e}\ (\text{refl}) \qquad \frac{\Gamma \vdash e_0 \leq_\tau e_1 \qquad \Gamma \vdash e_1 \leq_\tau e_2}{\Gamma \vdash e_0 \leq_\tau e_2}\ (\text{trans})$$

$$\frac{\Gamma, x:\tau' \vdash \mathscr{C}[x] : \tau \qquad \Gamma \vdash e_0 \leq_\tau e_1}{\Gamma \vdash \mathscr{C}[e_0] \leq_\tau \mathscr{C}[e_1]}\ (\text{mon})$$

$$\frac{}{0 + e =_{\mathsf{C}} e}\ (+\text{-idl}) \qquad \frac{}{e + 0 =_{\mathsf{C}} e}\ (+\text{-idr}) \qquad \frac{}{e_0 + (e_1 + e_2) =_{\mathsf{C}} (e_0 + e_1) + e_2}\ (+\text{-assoc})$$

$$\frac{}{\Gamma \vdash e_i \leq_\sigma \pi_i(e_0, e_1)}\ (\beta_\times)$$

$$\frac{}{\Gamma \vdash e_i \leq_\sigma \mathsf{case}\,\iota_i\,e\,\mathsf{of}\,\{(x:\sigma_i).e_i\}_{i=0,1}}\ (\beta_+)$$

$$\frac{}{\Gamma \vdash e\{e'/x\} \leq_\sigma (\lambda(x:\sigma').e)e'}\ (\beta_\to)$$

$$\frac{}{\Gamma \vdash e \leq_{F\{\vec{\sigma}/\vec{\alpha}\}[\delta]} \mathsf{d}_\delta\,(\mathsf{c}_\delta\,e)}\ (\beta_\delta)$$

$$\frac{}{\Gamma \vdash e\{(F[(y:\delta).\mathsf{fold}_\delta\,y\,\mathsf{of}\,x.e, e'])/x\} \leq_\sigma \mathsf{fold}_\delta\,(\mathsf{c}_\delta\,e')\,\mathsf{of}\,x.e}\ (\beta_{\delta\mathsf{fold}})$$

$$\frac{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}{\Gamma \vdash e \leq_{\tau\{\sigma/\alpha\}} (\Lambda\alpha.e)\,\sigma}\ (\beta_\forall)$$

Fig. 11: The size order relation that defines the semantics of the recurrence language. The macro $F[(y:\rho).e', e]$ is defined in Figure 12.

$$\begin{aligned}
t[(r:\rho).e', e] &= e'\{x \mapsto e\} \\
\tau_0[(y:\rho).e', e] &= e \\
(F_0 + F_1)[(y:\rho).e', e] &= \mathsf{case}\,e\,\mathsf{of}\,\{(x:F_i[\rho]).\iota_i\,(F_i[(y:\rho).e', x])\}_{i=0,1} \\
(F_0 \times F_1)[(y:\rho).e', e] &= (F_0[(y:\rho).e', \pi_0\,e], F_1[(y:\rho).e', \pi_1\,e]) \\
(\sigma_0 \to F)[(y:\rho).e', e] &= \lambda(x:\sigma_0).F[(y:\rho).e', e\,x]
\end{aligned}$$

Fig. 12: The macro $F[\rho; y.e', e]$.

description of size than what is started with. It may help to analogize with abstract interpretation here: an introductory form serves as an abstraction, whereas an elimination form serves as a concretization. In practice, the interpretation of products, sums, and arrows do not perform any abstraction, and so in the models we present in Section 6, the corresponding axioms are witnessed by equality (e.g., for $(\beta_\times)$, $[\![e_i]\!] = [\![\pi_i\,(e_0, e_1)]\!]$). That is not the case for datatypes and type quantification, so let us examine this in more detail.

Looking forward to definitions from Section 4, if σ is a source language type, then $\langle\!\langle\sigma\rangle\!\rangle$ is the *potential type* corresponding to σ and is intended to be interpreted as a set of sizes for σ values. It happens that $\langle\!\langle\sigma\ \mathtt{list}\rangle\!\rangle = \langle\!\langle\sigma\rangle\!\rangle\,\mathsf{list}$, so a $\sigma\ \mathtt{list}$ value $vs = [v_0, \ldots, v_{n-1}]$ is extracted as a list of $\langle\!\langle\sigma\rangle\!\rangle$ values, each of which represents the size of one of the v_is. Hence, a great deal of information is preserved about the original source-language program. But frequently the interpretation (the denotational semantics of the recurrence language) abstracts away many of those details. For example, we might interpret $\langle\!\langle\sigma\rangle\!\rangle\,\mathsf{list}$ as $\mathbf{N}$ (the natural numbers), nil as 0, and cons as successor (with respect to its second

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

argument), thereby yielding a semantics in which each list is interpreted as its length (we define two such "constructor-counting" models in Sections 7.2 and 7.3). Bearing in mind that $\langle\!\langle\sigma\rangle\!\rangle\ \mathsf{list} = \mu t.F$ with $F = \mathsf{unit} + \langle\!\langle\sigma\rangle\!\rangle \times t$, the interpretation of $F[\langle\!\langle\sigma\rangle\!\rangle\ \mathsf{list}]$ must be $[\![\mathsf{unit}]\!]+([\![\langle\!\langle\sigma\rangle\!\rangle]\!]\times\mathbf{N})$. Let us assume that $+$ and $\times$ are given their usual interpretations (though as we will see in Section 7, that is often not sufficient). For brevity, we will write c and d for $\mathsf{c}_{\langle\!\langle\sigma\rangle\!\rangle\ \mathsf{list}}$ and $\mathsf{d}_{\langle\!\langle\sigma\rangle\!\rangle\ \mathsf{list}}$. Thus, $\mathsf{c}(y, n) = 1 + n$ represents the size of a source-language list that is built using $\mathsf{c}_{\sigma\ \mathtt{list}}$ when applied to a head of size y (which is irrelevant) and a tail of size n. The question is, what should the value of $\mathsf{d}(1 + n)$ be? It ought to somehow describe all possible pairs that are mapped to $1 + n$ by c. Ignoring the possibility that it is an element of $[\![\mathsf{unit}]\!]$ (which seems obviously wrong), and assuming the $\mathbf{N}$-component ought to be n, no one pair $(y', n) \in [\![\langle\!\langle\sigma\rangle\!\rangle]\!]\times\mathbf{N}$ seems to do the job. However, if we assume the existence of an maximum element ∞ of $[\![\langle\!\langle\sigma\rangle\!\rangle]\!]$, then (∞, n) is an upper bound on all pairs (y', n) such that $\mathsf{c}(y', n) = 1 + n$, and so it seem reasonable to set $\mathsf{d}(1 + n) = (\infty, n)$. But in this case, the round trip is not an identity because $\mathsf{d}(\mathsf{c}(y, n)) = (\infty, n) \geq (y, n)$, and so (β_δ) and $(\beta_{\delta\mathsf{fold}})$ are not witnessed by equality.

Turning to type quantification, the standard interpretation of $\forall\alpha.\sigma$ is $\prod_{U\in\mathbf{U}} [\![\sigma]\!]\{\alpha \mapsto U\}$ for a suitable index set $\mathbf{U}$ (in the setting of predicative polymorphism, this does not pose any foundational difficulties), and the interpretation of a polymoprhic program is the $\mathbf{U}$-indexed tuple of all of its instances. Let $\lambda xs.e : \alpha\ \mathtt{list} \to \rho$ be a polymorphic program in the source language. The recurrence extracted from it essentially has the form $\Lambda\alpha.\lambda xs.E : \forall\alpha.\alpha\ \mathsf{list} \to \mathsf{C} \times \langle\!\langle\rho\rangle\!\rangle$. Let us consider a denotational semantics in which $[\![\sigma\ \mathsf{list}]\!] = \mathbf{N} \times \mathbf{N}$, where (k, n) describes a $\sigma\ \mathtt{list}$ value with maximum component size k and length n (this is a variant of the semantics in Section 7.3). We are then in the conceptually unfortunate situation that the analysis of this polymorphic recurrence depends on its instances, which are defined in terms of not only the list length, but also the sizes of the list values. Parametricity tells us that the list value sizes are irrelevant, but our model fails to convey that. Instead, we really want to interpret the type of the recurrence as $\mathbf{N} \to ([\![\mathsf{C}]\!]\times[\![\langle\!\langle\rho\rangle\!\rangle]\!])$, where the domain corresponds to list length. This is a non-standard interpretation of quantified types, and so the interpretations of quantifier introduction and elimination will also be non-standard. In our approach to solving this problem, those interpretations in turn depend on the existence of a Galois connection between $\mathbf{N}$ and $\mathbf{N} \times \mathbf{N}$, for example mapping the length n (quantified type) to (∞, n) (an upper bound on instances), and we might map (k, n) (instance type) to n. The round trip for type quantification corresponds to $(k, n) \mapsto n \mapsto (\infty, n)$, and hence $(\beta_\forall)$ is not witnessed by equality (we deploy the usual conjugation with these two functions in order to propogate the inequality to function types while respecting contravariance). We describe an instance of this sort of model construction in Section 7.4, although there we are not able to eliminate the $\mathbf{U}$-indexed product, and so the type of the recurrence is interpreted by $\prod_{U\in\mathbf{U}} \mathbf{N} \to ([\![\mathsf{C}]\!] \times [\![\langle\!\langle\rho\rangle\!\rangle]\!])$.

4 Recurrence extraction

A challenge in defining recurrence extraction is that computing only evaluation cost is insufficient for enabling compositionality because the cost of $f(g(x))$ depends on the size

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

of $g(x)$ as well as its cost. To drive this home, consider a typical higher order function such as

```
map = fn (f, xs) => fold (fn (x, r) => f x :: r) []
```

The cost of $\texttt{map}(f, xs)$ depends on the cost of evaluating f on the elements of xs, and hence (indirectly) on the sizes of the elements of xs. And since $\texttt{map}(f, xs)$ might itself be an argument to another function (e.g., another—map—), we also need to predict the sizes of the elements of $\texttt{map}(f, xs)$, which depends on the size of the output of f. Thus, to analyze —map—, we should be given a recurrence for the cost and size of $f(x)$ in terms of the size of x, from which we produce a recurrence that gives the cost and size of $\texttt{map}(f, xs)$ 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 (potential) uses of that value will cost (*use cost* would be another reasonable term for potential).

Motivated by this discussion, we define translations $\langle\!\langle \cdot \rangle\!\rangle$ from source language types to complexity types and $\| \cdot \|$ from source language terms to recurrence language terms so that if $e : \sigma$, then $\|e\| : \mathsf{C} \times \langle\!\langle \sigma \rangle\!\rangle$. In the recurrence language, we call an expression of type $\langle\!\langle \sigma \rangle\!\rangle$ a *potential* and an expression of type $\mathsf{C} \times \langle\!\langle \tau \rangle\!\rangle$ a *complexity*. We abbreviate $\mathsf{C} \times \langle\!\langle \tau \rangle\!\rangle$ by $\|\tau\|$. The first component of $\|e\|$ is intended to be an upper bound on the cost of evaluating e, and the second component of $\|e\|$ is intended to be an upper bound on the potential of e. The weakness of the size order axioms only allows us to conclude "upper bound" syntactically (hence the definition of the bounding relations in Figure 16), though one can define models of the recurrence language in which the interpretations are exact. The potential of a type-level 0 expression is a measure of the size of that value to which it evaluates, because that is how the value contributes to the cost of future computations. And as we just described, the potential of a type-$\rho \to \sigma$ function f is itself a function from potentials of type ρ (upper bounds on sizes of arguments x of f) to complexities of type σ (an upper bound on the cost of evaluating $f(x)$ and the size of the result).

Returning to $\texttt{map} : (\rho \to \sigma) \times \rho\ \texttt{list} \to \sigma\ \texttt{list}$, its potential should describe what future uses of map will cost, in terms of the potentials of its arguments. In this call-by-value setting, the arguments will already have been evaluated, so their costs do not play into the potential of map (the recurrence that is extracted from an application expression will take those costs into account). The above discussion suggests that $\langle\!\langle (\rho \to \sigma) \times \rho\ \texttt{list} \to \sigma\ \texttt{list} \rangle\!\rangle$ ought to be $(\langle\!\langle \rho \rangle\!\rangle \to \mathsf{C} \times \sigma) \times \langle\!\langle \rho\ \texttt{list} \rangle\!\rangle \to \mathsf{C} \times \langle\!\langle \sigma\ \texttt{list} \rangle\!\rangle$. For the argument function, we are provided a recurrence that maps ρ-potentials to σ-complexities. For the argument list, we are provided a $(\rho\ \texttt{list})$-potential. Using these, the potential of map must give the cost for doing the whole map and give a $(\sigma\ \texttt{list})$-potential for the value. This illustrates how the potential of a higher order function is itself a higher order function.

Since $\langle\!\langle \rho \rangle\!\rangle$ has as much "information" as ρ, syntactic recurrence extraction does not abstract values as sizes (e.g., we do not replace a list by its length). This permits us to prove a general bounding theorem independent of the particular abstraction (i.e., semantics) that a client may wish to use. Because of this, the complexity translation has a succinct description. For any monoid $(\mathsf{C}, +, 0)$, the writer monad (Wadler, 1992) $\mathsf{C} \times -$ is a monad with

$$\mathsf{return}(E) := (0, E)$$
$$E_1 \gg\!= E_2 := (\pi_0\, E_1 + \pi_0\, (E_2(\pi_1\, E_1)), \pi_1\, (E_2(\pi_2\, E_1)))$$

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\|\tau\| = \mathsf{C} \times \langle\!\langle \tau \rangle\!\rangle$$

$$\begin{aligned}
\langle\!\langle \alpha \rangle\!\rangle &= \alpha & \langle\!\langle \sigma\ \mathtt{susp} \rangle\!\rangle &= \|\sigma\| \\
\langle\!\langle \mathtt{unit} \rangle\!\rangle &= \mathsf{unit} & \langle\!\langle \mu t.F \rangle\!\rangle &= \mu t.\langle\!\langle F \rangle\!\rangle \\
\langle\!\langle \sigma_0 \times \sigma_1 \rangle\!\rangle &= \langle\!\langle \sigma_0 \rangle\!\rangle \times \langle\!\langle \sigma_1 \rangle\!\rangle & \langle\!\langle \forall \alpha.\tau \rangle\!\rangle &= \forall \alpha.\langle\!\langle \tau \rangle\!\rangle \\
\langle\!\langle \sigma_0 + \sigma_1 \rangle\!\rangle &= \langle\!\langle \sigma_0 \rangle\!\rangle + \langle\!\langle \sigma_1 \rangle\!\rangle \\
\langle\!\langle \sigma_0 \to \sigma_1 \rangle\!\rangle &= \langle\!\langle \sigma_0 \rangle\!\rangle \to \|\sigma_1\|
\end{aligned}$$

Fig. 13: The complexity and potential translation of types. Remember that although we have a grammar for structure functors F, they are actually just a subgrammar of the small types, so we do not require a separate translation function for them.

$\langle\!\langle \tau \rangle\!\rangle$	Potential translation of types.
$\|\tau\|$	Recurrence translation of types.
$\|e\|$	Recurrence extraction of expressions.
$\langle\!\langle \Gamma \rangle\!\rangle(x) = \langle\!\langle \Gamma(x) \rangle\!\rangle$	Potential translation of contexts.
E_c	$\pi_0\, E$ (cost component of E)
E_p	$\pi_1\, E$ (potential component of E)
$c +_c E$	$(c + E_c, E_p)$ ("adding cost")

Fig. 14: Notation related to recurrence language expressions and recurrence extraction.

The monad laws follow from the monoid laws for C. Thinking of 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 recurrence language, where source expressions that cost a step have the "effect" of incrementing the cost component, using the monad operation

$$\mathsf{incr}(E : \mathsf{C}) : \mathsf{C} \times \mathsf{unit} := (E, (\,)).$$

We write out the translation of types in Figure 13 and the recurrence extraction function explicitly in Figure 15. There is a certain amount of notation involved, which we summarize in Figure 14. Recurrence extraction is defined only for typeable terms and only for terms in the core language (Definition 2).

For an ordinary function type $\sigma_0 \to \sigma_1$, the translation $\langle\!\langle \sigma_0 \rangle\!\rangle \to \|\sigma_1\|$, i.e., $\langle\!\langle \sigma_0 \rangle\!\rangle \to \mathsf{C} \times \langle\!\langle \sigma_1 \rangle\!\rangle$ includes a cost component in the codomain. In contrast, a polymorphic function type $\forall \alpha.\tau$ is translated to $\forall \alpha.\langle\!\langle \tau \rangle\!\rangle$, which does not include a cost component. The reason for this discrepancy is that polymorphic functions in the source language are introduced by $\mathtt{let}\ x = e'\ \mathtt{in}\ e$, which evaluates e' to a value before binding x to a polymorphic version of that value. Thus, the elements of a polymorphic function type incur no immediate cost when they are instantiated (at an occurrence of a variable).

Our first order of business is to verify that recurrences extracted from terms in the source language are themselves typeable in the recurrence language. For a source-language context $\Gamma = x_0 : \tau_0, \ldots, x_{n-1} : \tau_{n-1}$, write $\langle\!\langle \Gamma \rangle\!\rangle$ for $x_0 : \langle\!\langle \tau_0 \rangle\!\rangle, \ldots, x_{n-1} : \langle\!\langle \tau_{n-1} \rangle\!\rangle$. For both the source and recurrence languages, we do not explicitly notate the free type variables of a

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$
\begin{aligned}
\|\Gamma, x : \forall\vec{\alpha}.\sigma \vdash x : \sigma\{\vec{\sigma}/\vec{\alpha}\}\| &= (0, x\,\langle\!\langle\vec{\sigma}\rangle\!\rangle) \\
\|\Gamma \vdash (\,) : \mathtt{unit}\| &= (0, (\,)) \\
\|\Gamma \vdash (e_0, e_1) : \sigma_0 \times \sigma_1\| &= (c_0 + c_1, (p_0, p_1)) \\
\|\Gamma \vdash \pi_i\, e : \sigma_i\| &= (c, \pi_i\, p) \\
\|\Gamma \vdash \iota_i\, e : \sigma_0 + \sigma_1\| &= (c, \iota_i\, p) \\
\|\Gamma \vdash \mathtt{case}\, e\, \mathtt{of}\, \{x.e_i\}_{i=0,1} : \sigma\| &= c +_c \mathsf{case}\, p\, \mathsf{of}\, \{(x : \langle\!\langle\sigma_i\rangle\!\rangle).\|e_i\|\}_{i=0,1} \\
&\qquad (\Gamma \vdash e : \sigma_0 + \sigma_1) \\
\|\Gamma \vdash \lambda x.e : \sigma' \to \sigma\| &= (0, \lambda(x : \langle\!\langle\sigma'\rangle\!\rangle).\|e\|) \\
\|\Gamma \vdash e_0\, e_1 : \sigma\| &= (c_0 + c_1) +_c p_0\, p_1 \\
\|\Gamma \vdash \mathtt{delay}(e) : \sigma\ \mathtt{susp}\| &= (0, \|e\|) \\
\|\Gamma \vdash \mathtt{force}(e) : \sigma\| &= c +_c p \\
\|\Gamma \vdash \mathtt{c}_\delta\, e : \delta\| &= (c, \mathtt{c}_{\langle\!\langle\delta\rangle\!\rangle}\, p) \\
\|\Gamma \vdash \mathtt{d}_\delta\, e : F[\delta]\| &= (c, \mathtt{d}_{\langle\!\langle\delta\rangle\!\rangle}\, p) \\
\|\Gamma \vdash \mathtt{fold}_\delta\, e'\, \mathtt{of}\, x.e : \sigma\| &= c' +_c \mathsf{fold}_{\langle\!\langle\delta\rangle\!\rangle}\, p'\, \mathsf{of}\, (x : \langle\!\langle F\rangle\!\rangle[\|\sigma\|]).1 +_c \|e\| \\
&\qquad (\delta = \mu t.F) \\
\|\Gamma \vdash \mathtt{let}\, x = e_0\, \mathtt{in}\, e_1 : \sigma_1\| &= c_0 +_c \|e_1\|\{\Lambda\vec{\alpha}.p_0/x\} \\
&\qquad (\Gamma, x : \forall\vec{\alpha}.\sigma_0 \vdash e_1 : \sigma_1)
\end{aligned}
$$

Fig. 15: The recurrence extraction function on source language terms. On the right-hand sides, $(c, p) = \|e\|$ and $(c_i, p_i) = \|e_i\|$ (note that $\|e\|$ is always a pair).

typing derivation. However, the intended invariant of the translation is that a source language derivation $\Gamma \vdash e : \tau$ with free type variables $\vec{\alpha}$ is translated to a recurrence language derivation $\langle\!\langle\Gamma\rangle\!\rangle \vdash \|e\| : \|\sigma\|$ that also has free type variables $\vec{\alpha}$.

Proposition 1 (Typeability of extracted recurrences)**.** *If $\Gamma \vdash e : \sigma$ is in the core language, then $\langle\!\langle\Gamma\rangle\!\rangle \vdash \|e\| : \|\sigma\|$.*

Proof See Appendix 2. ■

5 The bounding relation and the syntactic bounding theorem

We now turn to the bounding relation, which is a logical relation that is the main technical tool that relates source programs to recurrences. In this section, we will refer to source and recurrence language programs extensively, and so we will adopt the convention that E, E', etc. are metavariables for recurrence language terms. The bounding relation $e\theta \preceq_\sigma E$ is defined in Figure 16 and is intended to mean that E_c is a bound on the evaluation cost of $e\theta$ and E_p is a bound on the value to which $e\theta$ evaluates. Bounding of values is defined by an auxiliary relation $v \preceq^{\mathrm{val}}_\sigma E$. This latter relation morally should be defined by induction on σ, declaring that a value is bounded by a potential if its components are bounded by corresponding computations on that potential. Of course, function values are defined in terms

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

Expression bounding at open (monomorphic) types:

$$\frac{e\theta \downarrow^n v \qquad n \leq_{\mathsf{C}} E_c \qquad v \preceq^{\mathrm{val}}_{\sigma} E_p}{e\theta \preceq_{\sigma} E}$$

Value bounding at open (monomorphic) types:

$$\frac{\text{For all closed } \rho\colon v \preceq^{\mathrm{val}}_{\sigma\{\rho/\alpha\}} E\{\langle\!\langle\rho\rangle\!\rangle/\alpha\}}{v \preceq^{\mathrm{val}}_{\sigma} E}$$

Value bounding at closed (monomorphic) types:

$$\frac{}{() \preceq^{\mathrm{val}}_{\mathtt{unit}} E}$$

$$\frac{\{v_i \preceq^{\mathrm{val}}_{\sigma_i} \pi_i\, E\}_{i=0,1}}{(v_0, v_1) \preceq^{\mathrm{val}}_{\sigma_0\times\sigma_1} E} \qquad \frac{v \preceq^{\mathrm{val}}_{\sigma_i} E_i \qquad \iota_i\, E_i \leq_{\sigma_0+\sigma_1} E}{\iota_i\, v \preceq^{\mathrm{val}}_{\sigma_0+\sigma_1} E}$$

$$\frac{\{e\theta\{x \mapsto v'\} \preceq_{\sigma} E\, E' \mid v' \preceq^{\mathrm{val}}_{\rho} E'\}}{(\lambda x.e)\theta \preceq^{\mathrm{val}}_{\rho\to\sigma} E} \qquad \frac{e\theta \preceq_{\sigma} E}{(\mathtt{delay}\, e)\theta \preceq^{\mathrm{val}}_{\sigma\ \mathtt{susp}} E}$$

$$\frac{v \preceq^{\mathrm{val}}_{F,\delta} E' \qquad \mathsf{c}_{\langle\!\langle\delta\rangle\!\rangle} E' \leq_{\langle\!\langle\delta\rangle\!\rangle} E}{\mathsf{c}_{\delta}\, v \preceq^{\mathrm{val}}_{\delta} E}$$

Value bounding at type schemes:

$$\frac{\text{For all closed } \rho\colon v \preceq^{\mathrm{val}}_{\tau\{\rho/\alpha\}} E\{\langle\!\langle\rho\rangle\!\rangle/\alpha\}}{v \preceq^{\mathrm{val}}_{\forall\alpha.\tau} E}$$

Fig. 16: The type-indexed bounding relations.

of arbitrary expressions, and so $\preceq^{\mathrm{val}}_{\rho\to\sigma}$ must be defined in terms of $\preceq_{\sigma}$. The standard way to do so for a logical relation is to declare that $\lambda x.e$ is bounded as a value by E if whenever a value v' is bounded as a value by E', $e\{v'/x\}$ is bounded as an expression by $E\,E'$, and we adapt that same idea to our setting here. A naive approach to defining $\preceq^{\mathrm{val}}_{\delta}$ for $\delta = \mu t.F$ would have us define $v \preceq^{\mathrm{val}}_{\delta} E$ in a way that depends on $\preceq^{\mathrm{val}}_{F[\delta]}$, which is not a smaller type. If we did not permit arrows in shape functors, we could get around this by counting δ-constructors in v. Instead we must take a more general approach. In Figure 17, we define by induction on the structure function F the relations $\preceq_{F,\rho}$ and $\preceq^{\mathrm{val}}_{F,\rho}$ that correspond to bounding at type $F[\rho]$. We then define $\preceq^{\mathrm{val}}_{\mu t.F}$ in terms of $\preceq^{\mathrm{val}}_{F,\mu t.F}$.

The source language permits evaluation of closures with open type (in particular, when evaluating a `let`-binding), so the bounding relation is phrased in terms of open types. Value bounding at open type is defined in terms of all of its instances by closed monomorphic types—we do not enforce any parametricity properties here. Because source language

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\frac{e\theta \downarrow^n v \qquad n \preceq_{\mathsf{C}} E_c \qquad v \preceq^{\mathrm{val}}_{F,\rho} E_p}{e\theta \preceq_{F,\rho} E}$$

$$\frac{v \preceq^{\mathrm{val}}_{\rho} E}{v \preceq^{\mathrm{val}}_{t,\rho} E} \qquad \frac{v \preceq^{\mathrm{val}}_{\sigma} E}{v \preceq^{\mathrm{val}}_{\sigma,\rho} E}$$

$$\frac{\{v_i \preceq^{\mathrm{val}}_{F_i,\rho} \pi_i\, E\}_{i=0,1}}{(v_0, v_1) \preceq^{\mathrm{val}}_{F_0 \times F_1,\rho} E} \qquad \frac{v \preceq^{\mathrm{val}}_{F_i,\rho} E_i \qquad \iota_i\, E_i \preceq_{(F_0+F_1)[\rho]} E}{\iota_i\, v \preceq^{\mathrm{val}}_{F_0+F_1,\rho} E}$$

$$\frac{\{e\theta\{x \mapsto v'\} \preceq_{F,\rho} E\, E' \mid v' \preceq^{\mathrm{val}}_{\rho_0} E'\}}{(\lambda x.e)\theta \preceq^{\mathrm{val}}_{\rho_0 \to F,\rho} E}$$

Fig. 17: The shape-indexed bounding relations. When writing $v \preceq^{\mathrm{val}}_{F,\rho} E$, $\mathrm{ftv}(F) \subseteq \{t\}$ and ρ is closed.

type contexts assign type schemes to identifiers, the standard approach of extending a logical relation on closed terms to open terms by substituting related values requires us to also define a notion of value bounding at type schemes, and we again take this to be in terms of instances at closed types.

We present the relations as a formal derivation system of an inductive definition because the proofs of Lemmas 15 and 16 (technical lemmas needed for the proof of Theorem 5, the bounding theorem) rely on a well-founded notion of subderivation. A least relation closed under these rules (which contain a negative occurrence of the relation being defined in the $\to$ rule) exists because the type subscript gets smaller in all bounding premises ($\preceq$ or $\preceq^{\mathrm{val}}$, not $\preceq$, which is the previously defined size relation on recurrence language terms). The premise types are smaller for an ordering that considers all substitution instances $\tau\{\rho/\alpha\}$ of τ with a closed monomorphic type ρ to be smaller than the polymorphic type $\forall\alpha.\tau$ or a type with a free variable $\alpha.\tau$; this ordering is sufficient because of the restriction to predicative polymorphism. Although the derivations are infinitary as a result of the clauses corresponding to arrow types and shapes, it is straightforward to assign ordinal ranks to derivations so that the rank of any derivation is strictly larger than the rank of any of its immediate subderivations, justifying such a proof by induction on derivations.

The (value) bounding relations in Figures 16 and 17 are really defined on typing derivations. That is, we really define the relations

$$(\_ \vdash e\theta : \sigma) \preceq_{\sigma} (\_ \vdash E : \|\sigma\|) \qquad (\_ \vdash e\theta : F[\rho]) \preceq_{F,\rho} (\_ \vdash E : \|F[\rho]\|)$$
$$(\_ \vdash v : \sigma) \preceq^{\mathrm{val}}_{\sigma} (\_ \vdash E : \langle\!\langle \sigma \rangle\!\rangle) \qquad (\_ \vdash v : F[\rho]) \preceq^{\mathrm{val}}_{F,\rho} (\_ \vdash E : \langle\!\langle F[\rho] \rangle\!\rangle)$$

The following lemma acts as an inversion theorem for the bounding relation at inductive types.

Lemma 2.

1. *If* $e\theta \preceq_{F[\rho]} E$, *then* $e\theta \preceq_{F,\rho} E$.
2. *If* $e\theta \preceq^{\mathrm{val}}_{F[\rho]} E$, *then* $e\theta \preceq^{\mathrm{val}}_{F,\rho} E$.

Proof (2) implies (1), so it suffices to prove the latter, which is done by a straightforward induction on shape functors. ■

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

The bounding relations on closures are extended to (open) terms in the standard way for logical relations.

Definition (Bounding relation)**.**

1. *Let θ be a Γ-environment and Θ a $\langle\!\langle\Gamma\rangle\!\rangle$-environment. We write $\theta \preceq^{\mathrm{val}}_{\Gamma} \Theta$ to mean that for all $x \in \mathrm{dom}\,\Gamma$, $\theta(x) \preceq^{\mathrm{val}}_{\Gamma(x)} \Theta(x)$ (note that $\Gamma(x)$ is a type scheme, so this relation is value bounding at a type scheme).*
2. *We write $(\Gamma \vdash e : \sigma) \preceq_{\sigma} (\langle\!\langle\Gamma\rangle\!\rangle \vdash E : \|\sigma\|)$ to mean that for all $\theta \preceq^{\mathrm{val}}_{\Gamma} \Theta$, $e\theta \preceq_{\sigma} E\{\Theta\}$.*

The syntactic bounding theorem relies on various weakening and substitution properties that we collect here.

Lemma 3 (Weakening)**.**

1. *If $e \preceq E$ and $E \leq E'$, then $e \preceq E'$.*
2. *If $v \preceq^{\mathrm{val}} E$ and $E \leq E'$, then $v \preceq^{\mathrm{val}} E'$.*

Lemma 4. *$c + E_c \leq (c +_c E)_c$ and $E_p \leq (c +_c E)_p$. In particular, if $e\theta \downarrow^n v$, $n \leq c + E_c$, and $v \preceq^{\mathrm{val}} E_p$, then $e\theta \preceq c +_c E$.*

The main theorem is analogous to the fundamental theorem for any logical relation: every source language program is related (bounded by) the syntactic recurrence extracted from it. The proof is somewhat technically involved, but at its core follows the reasoning typical in the proof of any such fundamental theorem, so we delegate it to the Appendix.

Theorem 5 (Syntactic bounding theorem)**.** *If e is in the core language and $\Gamma \vdash e : \sigma$, then $e \preceq_{\sigma} \|e\|$.*

Proof See Appendix 3. ■

6 Environment models

The syntactic bounding theorem tells us that the syntactic recurrences extracted from source programs provide bounds on the evaluation cost and potential of those programs. However, the syntactic recurrences maintain sufficient information about the source program to describe cost and potential in terms of almost any notion of size. In particular, a syntactic recurrence extracted from a program over an inductive type maintains all the structure of the values of that type—e.g., a syntactic recurrence over a list program describes the bounds in terms of lists again. It is by defining a denotational semantics for the recurrence language that we obtain a "traditional" recurrence because that permits us to abstract inductive values to some notion of size. We might define a semantics in which a σ tree type is interpreted by the natural numbers $\mathbf{N}$, with the constructor interpreted in terms of either the maximum function (so a tree is interpreted by its height) or the sum function (so a tree is interpreted by its size). So a semantic value in $\mathsf{unit} + \sigma \times \mathbf{N} \times \mathbf{N}$, the

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

one-step unfolding of the interpretation of the tree type, tells us the sizes of the data supplied to the tree constructor. The constructor tells us the size of the tree constructed from such data, and the destructor tells us about the kind of data that can be used to construct a tree of a given size. The denotation of the recurrence extracted from a source program f is then a function T such that $T(n)$ is (a bound on) the cost and size of the result of $f(x)$ when x has size at most n. In other words, the end goal is a "semantic" recurrence obtained by composing a denotation function with the extraction function. Soundness of the denotational semantics with respect to the size ordering in conjunction with the syntactic bounding theorem ensures that the semantic recurrence also provides bounds on the cost and potential of the source program in terms of the potentials of its arguments.

To that end, we need to define an appropriate notion of model for the recurrence language. We will define environment (Henkin) models following (Mitchell, 1996, Ch. 9.2.4), which in turn follows Bruce *et al*. (1990), specializing the definition to the setting of the recurrence language. Since the recurrence language is characterized by the size order, we require that types be interpreted by preorders, and what would usually be equations describing various semantic functions will be inequalities. This leads to a slight challenge in extending an interpretation of inductive-type constructors and destructors to a canonical interpretation of fold_δ because the interpretation of δ is no longer an initial algebra. However, we shall see that it is sufficient to have a initiality condition that is weak (requires existence, but not uniqueness) and lax (is an inequality, not an equality), and that we can arrange.

Applicative structures (and hence premodels and models) are defined in terms of preordered sets. In such a setting, it is natural to restrict ourselves to functions that respect the pre-order structure—i.e., monotone functions. So in the remaining sections, when A and B are preordered sets, we write $A \to B$ for the set of monotone functions from A to B, and $A \rightharpoonup B$ for the set of partial monotone functions from A to B. $A \to B$ is preordered pointwise, and $\mathrm{id}_A : A \to A$ is the identity function (we drop the subscript when clear from context). We also frequently write $\lambda\!\!\lambda a. \cdots$ for the semantic function that takes a to $\cdots$.

6.1 Models of the recurrence language

We start by defining the notions of type frame and applicative structure for the recurrence language.

Definition. *A* type frame *is specified by the following data:*

- *A set* $\mathbf{U}_{sm}$ *of* small semantic types *and a set* $\mathbf{U}_{lg}$ *of* large semantic types *with* $\mathbf{U}_{sm} \subseteq \mathbf{U}_{lg}$*;*
- *Distinguished semantic types* $U_{\mathsf{C}}, U_{\mathsf{unit}} \in \mathbf{U}_{sm}$*;*
- *Functions* $\underline{\times} : \mathbf{U}_{sm} \times \mathbf{U}_{sm} \to \mathbf{U}_{sm}$, $\underline{+} : \mathbf{U}_{sm} \times \mathbf{U}_{sm} \to \mathbf{U}_{sm}$, $\underline{\to} : \mathbf{U}_{sm} \times \mathbf{U}_{sm} \to \mathbf{U}_{sm}$, *and* $\underline{\mu} : (\mathbf{U}_{sm} \to \mathbf{U}_{sm}) \rightharpoonup \mathbf{U}_{sm}$*; and*
- *A function* $\underline{\forall} : (\mathbf{U}_{sm} \to \mathbf{U}_{lg}) \rightharpoonup \mathbf{U}_{lg}$.

Let TyVar be the set of type variables and let $\eta : \mathrm{TyVar} \to \mathbf{U}_{sm}$. The denotation of τ with respect to η, $[\![\tau]\!]\eta$, is given in Figure 18.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\begin{aligned} [\![\sigma \times \sigma']\!]\eta &= [\![\sigma]\!]\eta \mathbin{\underline{\times}} [\![\sigma']\!]\eta \\ [\![\alpha]\!]\eta = \eta(\alpha) \qquad [\![\sigma + \sigma']\!]\eta &= [\![\sigma]\!]\eta \mathbin{\underline{+}} [\![\sigma']\!]\eta \\ [\![\mathsf{unit}]\!]\eta = U_{\mathsf{unit}} \qquad [\![\sigma \to \sigma']\!]\eta &= [\![\sigma]\!]\eta \mathbin{\underline{\to}} [\![\sigma']\!]\eta \\ [\![\mu t.F]\!]\eta &= \underline{\mu}(\lambda\!\!\lambda V.[\![F]\!]\eta\{t \mapsto V\}) \end{aligned}$$

$$[\![\forall\alpha.\tau]\!]\eta = \underline{\forall}(\lambda\!\!\lambda U.[\![\tau]\!]\eta\{\alpha \mapsto U\})$$

Fig. 18: The denotation (partial) function of types and type schemes into a type frame.

Definition. *A type frame is a* type model *if for all F and η, $\lambda\!\!\lambda V.[\![F]\!]\eta\{t \mapsto V\} \in \operatorname{dom} \underline{\mu}$ and for all τ and all η, $\lambda\!\!\lambda U.[\![\tau]\!]\eta\{\alpha \mapsto U\} \in \operatorname{dom} \underline{\forall}$ (and hence $[\![\tau]\!]\eta$ is defined for all τ and η).*

Definition. *An* applicative structure *is specified by the following data:*

- *A type model* $(\mathbf{U}_{sm}, \mathbf{U}_{lg})$.
- *For each* $U \in \mathbf{U}_{lg}$*, a preordered set* $(D^U, \leq_U)$.
- *For each* $\Phi \in \operatorname{dom} \underline{\mu}$ *and* $U, V \in \mathbf{U}_{sm}$*, a function* $\Phi_{U,V} : (D^U \to D^V) \to (D^{\Phi\, U} \to D^{\Phi\, V})$.
- *Distinguished elements* $0, 1 \in D^{U\mathsf{C}}$ *and an associative function* $+ : D^{U\mathsf{C}} \to D^{U\mathsf{C}}$ *such that* 0 *is a right- and left identity for* $+$.
- *A distinguished element* $* \in D^{U_{\mathsf{unit}}}$
- *For each* $U, V \in \mathbf{U}_{sm}$*, functions*

  $$(D^U \to D^V) \xrightarrow{\underline{\mathrm{Abs}}_{U,V}} D^{U \mathbin{\underline{\to}} V} \xrightarrow{\underline{\mathrm{App}}_{U,V}} (D^U \to D^V)$$

  such that $\underline{\mathrm{App}} \circ \underline{\mathrm{Abs}} \geq \mathrm{id}$*. Note that* $\underline{\mathrm{Abs}}$ *is a partial function.*
- *For each* $U_0, U_1 \in \mathbf{U}_{sm}$*, functions*

  $$D^{U_0} \times D^{U_1} \xrightarrow{\underline{\mathrm{Pair}}_{U_0,U_1}} D^{U_0 \mathbin{\underline{\times}} U_1} \xrightarrow{\underline{\mathrm{Proj}}^i_{U_0,U_1}} D^{U_i}$$

  such that $\underline{\mathrm{Proj}}^i(\underline{\mathrm{Pair}}(a_0, a_1)) \geq a_i$.
- *For each* U_0*,* U_1*,* $V \in \mathbf{U}_{sm}$*, functions*

  $$D^{U_i} \xrightarrow{\underline{\mathrm{Inj}}^i_{U_0,U_1}} D^{U_0 \mathbin{\underline{+}} U_1} \xrightarrow{\underline{\mathrm{Case}}_{U_0,U_1,V}} (D^{U_0} \to V) \times (D^{U_1} \to V) \to D^V$$

  such that $(\underline{\mathrm{Case}} \circ \underline{\mathrm{Inj}}^i)\, a\, (f_0, f_1) \geq f_i\, a$*. We often write* $\underline{\mathrm{Case}}(a, f_0, f_1)$ *for* $\underline{\mathrm{Case}}\, a\, (f_0, f_1)$.
- *For each* $\Phi \in \operatorname{dom} \underline{\mu}$*, functions*

  $$D^{\Phi(\underline{\mu}\,\Phi)} \xrightarrow{\underline{\mathrm{C}}_\Phi} D^{\underline{\mu}\,\Phi} \xrightarrow{\underline{\mathrm{D}}_\Phi} D^{\Phi(\underline{\mu}\,\Phi)}$$

  such that $\underline{\mathrm{D}} \circ \underline{\mathrm{C}} \geq \mathrm{id}$.
- *For each* $\Phi \in \operatorname{dom} \underline{\mu}$ *and* $U \in \mathbf{U}_{sm}$*, functions* $\underline{\mathrm{Fold}}_{\Phi,U} : (D^{\Phi\, U} \to D^U) \to (D^{\underline{\mu}\,\Phi} \to D^U)$ *such that* $(\underline{\mathrm{Fold}}_{\Phi,U} f) \circ \underline{\mathrm{C}}_\Phi \geq f \circ (\Phi_{\underline{\mu}\,\Phi,U}(\underline{\mathrm{Fold}}_{\Phi,U} f))$.
- *For each* $\Phi \in \operatorname{dom} \forall$*, functions*

  $$\prod_{U \in \mathbf{U}_{sm}} D^{\Phi(U)} \xrightarrow{\underline{\mathrm{TyAbs}}_\Phi} D^{\underline{\forall}(\Phi)} \xrightarrow{\underline{\mathrm{TyApp}}_\Phi} \prod_{U \in \mathbf{U}_{sm}} D^{\Phi(U)}$$

  such that $\underline{\mathrm{TyApp}} \circ \underline{\mathrm{TyAbs}}) \geq \mathrm{id}$*. Note that* $\underline{\mathrm{TyAbs}}$ *is a partial function.*

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$
\begin{aligned}
[\![\Gamma, x:\tau \vdash x:\tau]\!]\eta &= \eta(x)\\
[\![\Gamma \vdash ():\mathsf{unit}]\!]\eta &= *\\
[\![\Gamma \vdash (e_0,e_1):\sigma_0\times\sigma_1]\!]\eta &= \underline{\mathrm{Pair}}([\![e_0]\!]\eta,[\![e_1]\!]\eta)\\
[\![\Gamma \vdash \pi_i\, e:\sigma_i]\!]\eta &= \underline{\mathrm{Proj}}^i([\![e]\!]\eta)\\
[\![\Gamma \vdash \iota_i\, e:\sigma_0+\sigma_1]\!]\eta &= \underline{\mathrm{Inj}}^i([\![e]\!]\eta)\\
[\![\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ x.e_0; x.e_1:\sigma]\!]\eta &= \underline{\mathrm{Case}}([\![e]\!]\eta, \lambda\!\!\lambda a.[\![e_0]\!]\eta\{x\mapsto a\}, \lambda\!\!\lambda a.[\![e_1]\!]\eta\{x\mapsto a\})\\
[\![\Gamma \vdash \lambda x.e:\sigma\to\sigma']\!]\eta &= \underline{\mathrm{Abs}}(\lambda\!\!\lambda a.[\![e]\!]\eta\{x\mapsto a\})\\
[\![\Gamma \vdash e\ e':\sigma]\!]\eta &= \underline{\mathrm{App}}([\![e]\!]\eta)\,([\![e']\!]\eta)\\
[\![\Gamma \vdash \mathsf{c}_\delta\, e:\delta]\!]\eta &= \underline{\mathrm{C}}_{\boldsymbol{\lambda} V.[\![F]\!]\eta\{t\mapsto V\}}([\![e]\!]\eta)\\
[\![\Gamma \vdash \mathsf{d}_\delta\, e:F[\delta]]\!]\eta &= \underline{\mathrm{D}}_{\boldsymbol{\lambda} V.[\![F]\!]\eta\{t\mapsto V\}}([\![e]\!]\eta)\\
[\![\Gamma \vdash \mathsf{fold}_\delta\, e'\ \mathsf{of}\ x.e:\sigma]\!]\eta &= \underline{\mathrm{Fold}}_{\boldsymbol{\lambda} V.[\![F]\!]\eta\{t\mapsto V\},[\![\sigma]\!]\eta}(\lambda\!\!\lambda a.[\![e]\!]\eta\{x\mapsto a\})\,([\![e']\!]\eta)\\
[\![\Gamma \vdash \Lambda\alpha.e:\forall\alpha.\tau]\!]\eta &= \underline{\mathrm{TyAbs}}_{\boldsymbol{\lambda} U.[\![\tau]\!]\eta\{\alpha\mapsto U\}}(\lambda\!\!\lambda U.[\![\Gamma \vdash e:\tau]\!]\eta\{\alpha\mapsto U\})\\
[\![\Gamma \vdash e\,\sigma:\tau\{\sigma/\alpha\}]\!]\eta &= \underline{\mathrm{TyApp}}_{\boldsymbol{\lambda} U.[\![\tau]\!]\eta\{\alpha\mapsto U\}}([\![\Gamma \vdash e:\forall\alpha.\tau]\!]\eta)([\![\sigma]\!]\eta)
\end{aligned}
$$

Fig. 19: The denotation (partial) function into an applicative structure. For constructors and destructors, assume $\delta = \mu t.F$ and $\mathrm{fv}(\delta) = \{\alpha_0,\ldots,\alpha_{n-1}\}$, and define $\eta^* = \eta\{\vec{\alpha}\mapsto\vec{U}\}$.

Remember that when we write, e.g., $D^U \to D^V$, we mean the monotone functions from D^U to D^V, and hence all of the semantic functions that make up the data of an applicative structure are monotone.

We write $\mathbf{U} = (\mathbf{U}_{sm}, \mathbf{U}_{lg}, \{D^U\}_{U\in\mathbf{U}_{lg}})$ for a typical applicative structure, or just $\mathbf{U} = \{D^U\}_{U\in\mathbf{U}_{lg}}$ when $\mathbf{U}_{lg}$ is clear from context. For a context Γ define $\mathrm{tyvar}(\Gamma) = \{\alpha \mid \alpha \text{ occurs in } \mathrm{ftv}(\Gamma(x)) \text{ for some } x\}$. Define a Γ-*environment* to be a map η such that

- $\eta(\alpha)\in\mathbf{U}_{sm}$ for $\alpha\in\mathrm{tyvar}(\Gamma)$; and
- $\eta(x)\in D^{[\![\Gamma(x)]\!]\eta}$ for $x\in\mathrm{dom}\,\Gamma$.

For an applicative structure and environment η, define a partial denotation function $[\![\Gamma\vdash e:\sigma]\!]\eta$ as in Figure 19. The only way in which $[\![\cdot]\!]\cdot$ may fail to be total is if the arguments to $\underline{\mathrm{Abs}}$ or $\underline{\mathrm{TyAbs}}$ are not in the corresponding domains (because we start with a type model, we know that $\underline{\mu}$ and $\underline{\forall}$ are only applied to functions in their domains).

Definition. *Let* $\mathbf{U}$ *be an applicative structure.*

1. $\mathbf{U}$ *is a* premodel *if*
   - *Whenever* $\Gamma\vdash e:\tau$ *and* η *is a* Γ*-environment,* $[\![\Gamma\vdash e:\tau]\!]\eta$ *is defined and an element of* $D^{[\![\tau]\!]\eta}$*; and*
   - *Whenever* $\Gamma, y:\rho\vdash e':\sigma$ *and* $\Gamma\vdash e:F[\rho]$ *and* η *is a* Γ*-environment,*

     $$[\![F[(y:\rho).e', e]]\!]\eta \le_{[\![\sigma]\!]\eta} (\lambda\!\!\lambda V.[\![F]\!]\eta\{t\mapsto V\})_{[\![\rho]\!]\eta,[\![\sigma]\!]\eta}(\lambda\!\!\lambda a.[\![e']\!]\eta\{y\mapsto a\})([\![e]\!]\eta).$$

2. $\mathbf{U}$ *is a* model *if* $\mathbf{U}$ *is a premodel and whenever* $\Gamma\vdash e\le_\tau e'$*, and* η *is a* Γ*-environment,* $[\![\Gamma\vdash e:\sigma]\!]\eta \le_{[\![\tau]\!]\eta} [\![\Gamma\vdash e':\sigma]\!]\eta$.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

The indirection of interpreting syntactic types by semantic types, and then interpreting terms of a given syntactic type as elements of a domain associated to the corresponding semantic type is necessary, especially in our setting of nonstandard models. This makes is much easier (seemingly, *possible*) to define things like the $\underline{\mu}$ operator. Without the indirection, we would have to define $\underline{\mu}$ on (functions on) a collection of domains, some of which represent syntactic types. That ends up being very difficult to do. For example, we might have to first define a notion of polynomial function on the semantic domains in order to define the domain of $\underline{\mu}$, and then somehow identify each semantic polynomial function with a structure functor. But doing so gets us into problems with unique representation; e.g., there may be multiple structure functors corresponding to the same semantic polynomial. And with nonstandard models, we seem to have even more troubles because we end up trying to define the interpretations of inductive types simultaneously with the $\underline{\mu}$ function. But first interpreting the syntactic types by semantic types gives us a way around these problems because (if we wish) we can define the semantic types to be closely tied to the syntactic types. That is exactly what we do for the standard type frame, so we can essentially define $\underline{\mu}$ syntactically, and then choose a domain corresponding to $\mu t.F$ (which is a semantic type as well as a syntactic one) after having defined $\underline{\mu}$.

Lemma 6. *Let* **U** *be a premodel. Then:*

1. $[\![\tau\{\sigma/\alpha\}]\!]\eta = [\![\tau]\!]\eta\{\alpha \mapsto [\![\sigma]\!]\eta\}$. *If* $\alpha \notin \mathrm{ftv}(\tau)$ *then for all* U, $[\![\tau]\!]\eta = [\![\tau]\!]\eta\{\alpha \mapsto U\}$ *and for all term variables* x *and all* a, $[\![\tau]\!]\eta = [\![\tau]\!]\eta\{x \mapsto a\}$.
2. $[\![e\{e'/x\}]\!]\eta = [\![e]\!]\eta\{x \mapsto [\![e']\!]\eta\}$. *If* $x \notin \mathrm{fv}(e)$, *then for all* a, $[\![e]\!]\eta = [\![e]\!]\eta\{x \mapsto a\}$.
3. *If* $a \leq a'$, *then* $[\![e]\!]\eta\{x \mapsto a\} \leq [\![e]\!]\eta\{x \mapsto a'\}$; *in other words,* $\lambda\!\!\lambda a.[\![e]\!]\eta\{x \mapsto a\}$ *is monotone.*

Proposition 2 (Environment model soundness)**.** *If* **U** *is an premodel, then* **U** *is a model.*

Proof By induction on the derivation of $\Gamma \vdash e \leq_\tau e'$. ■

One might hope that a model of the fragment of the recurrence language that omits fold_δ can be extended to one that does, but in our setting this does not quite hold. Since we only have directed versions of the usual equalities, initial algebras for structure functors may not exist. And even if they do, they are not necessarily what we want. For clarity, in this discussion, we will write syntactic types for semantic types. The point behind different semantics is to abstract inductive values to some notion of size, and when this abstraction is non-trivial, $D^{\mu t.F}$ and $D^{F[\mu t.F]}$ are probably not isomorphic. Instead of the usual initial algebra for interpreting $\mu t.F$, we typically want an algebra $\underline{\mathrm{C}}_F : D^{F[\mu t.F]} \to D^{\mu t.F}$ such that for any other algebra $s : D^{F[\sigma]} \to D^\sigma$, there is a function $\underline{\mathrm{Fold}}_{F,\sigma}\ s$ that makes the diagram

$$\begin{array}{ccc} D^{F[\mu t.F]} & \xrightarrow{\ \underline{\mathrm{C}}_F\ } & D^{\mu t.F} \\ \Big\downarrow{\scriptstyle \underline{\mathrm{Map}}_F(\underline{\mathrm{Fold}}_F\ s)} & \leq & \Big\downarrow{\scriptstyle \underline{\mathrm{Fold}}_F\ s} \\ D^{F[\sigma]} & \xrightarrow{\ s\ } & D^\sigma \end{array}$$

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

commute, where $\underline{\mathrm{Map}}_F$ is a semantic function that corresponds to the $F[\cdot,\cdot]$ macro. Relative to the usual definition of initial algebra, this requirement is *weak*, in that we ask only for existence of a $\underline{\mathrm{Fold}}$ function making the diagram commute (β reduction) and not the uniqueness of $\underline{\mathrm{Fold}}$ (η/induction), and it is *lax*, in that we ask that β-reduction holds only as an inequality, rather than an equality.

Nonetheless, under assumptions that turn out to be relatively easy to ensure, we can define $\underline{\mathrm{Fold}}_{\Phi,U}$. Given a subset $X \subseteq A$ of a preordered set A, we say that $a \in A$ is a *least upper bound* of X, written $a = \bigvee X$, if for all $x \in X$, $x \leq a$, and if $b \in A$ satisfies the condition that for all $x \in X$, $x \leq b$, then $a \leq b$. When A is preordered, $\bigvee X$ may not exist, and when it does, it need not be unique. If A is a *partial* order (i.e., $a \leq b \leq a$ implies that $a = b$), then $\bigvee X$ is unique when it exists, and we say that A is a *complete upper semi-lattice* if A is a partial order and $\bigvee X$ exists for every $X \subseteq A$. Though this seems like a very strong condition, in practice it is easy to ensure.

In a model in which every D^U is a complete upper semi-lattice, we would like to define

$$\underline{\mathrm{Fold}}_{\Phi,U}\, s\, x = \bigvee \{ s\big(\Phi_{\underline{\mu}\,\Phi,U}(\underline{\mathrm{Fold}}_{\Phi,U} s)\, z\big) \mid z \in D^{\Phi(\underline{\mu}\,\Phi)}, \underline{\mathrm{C}}_\Phi\, z \leq x \}. \tag{std-fold}$$

A priori, this definition may not be well founded, but in fact it is, as shown in the next proposition.

Proposition 3. *Suppose that* $\mathbf{U} = \{D^U\}_{U \in \mathbf{U}_{lg}}$ *is a model of the fragment of the recurrence language that omits* fold_δ *and each* D^U *is a complete upper semi-lattice, and suppose that* $\underline{\mathrm{Fold}}$ *is defined by* (std-fold)*. Then:*

1. *For all* s, $\underline{\mathrm{Fold}}_{\Phi,U}\, s$ *is total and monotone.*
2. $\underline{\mathrm{Fold}}_{\Phi,U}$ *is total and monotone.*

Proof

1. Fix s and consider
$$Q = \lambda g.\lambda x. \bigvee \{ s\big(\Phi_{\underline{\mu}\,\Phi,U}\, g\, z\big) \mid z \in D^{\Phi(\underline{\mu}\,\Phi)}, \underline{\mathrm{C}}_\Phi\, z \leq x \}.$$
$Q : (D^{\underline{\mu}\,\Phi} \to D^U) \to (D^{\underline{\mu}\,\Phi} \to D^U)$ and it is easy to see that Q is monotone. Since D^U is a complete upper semi-lattice, $D^{\underline{\mu}\,\Phi} \to D^U$ is a complete partial order. So Q has a least fixed point; that is, $\underline{\mathrm{Fold}}_{\Phi,U}\, s$.[4] Monotonicity of $\underline{\mathrm{Fold}}_{\Phi,U}\, s$ is immediate from its definition.
2. Totality follows from (1) and monotonicity from the fact that the function that maps a monotone function to its least fixed point is itself monotone.

■

The proof of Proposition 3, and hence the interpretation of fold_δ, may seem a bit heavy-handed, making use of general least fixed point theorems and even iterating into the transfinite. As we noted earlier, we are in a setting in which we do not have (and do not

[4] The least fixed point is obtained by the standard iteration of Q starting at the bottom element. Because we only have that Q is monotone (not necessarily continuous on chains), the iteration may have to be extended transfinitely—see Davey & Priestley (1999, Exercise 8.19).

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

want) initial algebras, but must nonetheless show an initiality-like property of a given algebra. Accordingly, we would expect to use technology at least as strong as that needed for typical initial algebra existence theorems. The canonical such theorem (e.g., as described by Aczel, 1988, Thm. 7.6) verifies that the least fixed point of a set-continuous operator is an initial algebra, and the verification consists of constructing the equivalent of $\underline{\text{Fold}}\ s$ by induction on the (ordinal-indexed) construction of the least fixed point.

The reader may have noticed that an alternative possible definition for $\underline{\text{Fold}}$ is

$$\underline{\text{Fold}}\ s\ x = s(\Phi(\underline{\text{Fold}}\ s)(\underline{\text{D}}\ x))$$

and Proposition 3 would still hold. This fact witnesses that the initiality condition for $\underline{\text{C}}_\Phi : D^{\Phi(\underline{\mu}\,\Phi)} \to D^{\underline{\mu}\,\Phi}$ is weak, in that functions to other algebras are not unique. In practice, this alternative definition yields far worse bounds for extracted recurrences, because we end up defining $\underline{\text{D}}_\Phi\ x = \bigvee\{z \mid \underline{\text{C}}_\Phi(z) \leq x\}$ and so $\underline{\text{Fold}}\ s\ x = s(\Phi(\underline{\text{Fold}}\ s)(\bigvee\{z \mid \underline{\text{C}}_\Phi(z) \leq x\}))$. Monotonicity of f only allows us to conclude that $f(\bigvee X) \geq \bigvee\{f(x) \mid x \in X\}$, but this putative definition for $\underline{\text{Fold}}\ s$ exposes a case in which this inequality is strict.

6.2 The standard type frame

Our last step in our general discussion of models is to define the type frame upon which all of our examples will be based. It gives us enough data to provide a standard definition of the functions $\Phi_{U,V}$, which in turn lets us use (std-fold) to define $\underline{\text{Fold}}$ and so for most of our examples, it will suffice to define $\underline{\text{C}}_\Phi$ (because we will set $\underline{\text{D}}_\Phi = \bigvee\{z \mid \underline{\text{C}}_\Phi(z) \leq x\}$). Our examples are all based on variations of the *standard type frame*, which is defined as follows:

- $\mathbf{U}_{sm}$ is the set of closed types and $\mathbf{U}_{lg}$ the set of closed type schemes of the recurrence language.
- $\underline{\rightarrow}$, $\underline{\times}$, and $\underline{+}$ are the standard type constructors; e.g., $\sigma_0 \mathbin{\underline{+}} \sigma_1 = \sigma_0 + \sigma_1$.
- $\text{dom}\,\underline{\mu} = \{\lambda\!\!\lambda\sigma.F[\sigma] \mid \text{fv}(F) \subseteq \{t\}\}$ and $\underline{\mu}(\lambda\!\!\lambda\sigma.F[\sigma]) = \mu t.F$ (we call a structure functor F with $\text{fv}(F) \subseteq \{t\}$ *closed*).
- $\text{dom}\,\underline{\forall} = \{\lambda\!\!\lambda\sigma.\tau\{\sigma/\alpha\} \mid \text{fv}(\tau) = \{\alpha\}\}$ and $\underline{\forall}(\lambda\!\!\lambda\sigma.\tau\{\sigma/\alpha\}) = \forall\alpha.\tau$.

It is straightforward to show that if $\lambda\!\!\lambda\sigma.F[\sigma] = \lambda\!\!\lambda\sigma.F'[\sigma]$, then $F = F'$, and if $\lambda\!\!\lambda\sigma.\tau\{\sigma/\alpha\} = \lambda\!\!\lambda\sigma.\tau'\{\sigma/\alpha\}$, then $\tau = \tau'$, so $\underline{\mu}$ and $\underline{\forall}$ are well defined. It should be clear and occasionally helpful to observe that for any τ and environment $\eta = \{\alpha_0 \mapsto \sigma_0, \ldots, \alpha_{n-1} \mapsto \sigma_{n-1}\}$, $[\![\tau]\!]\eta = \tau\{\vec{\sigma}/\vec{\alpha}\}$. For models based on the standard type frame and any closed structure functor F, we will usually write F in place of $\lambda\!\!\lambda\sigma.F[\sigma]$ in subscripts for readability.

Proposition 4. *The standard type frame is a type model.*

For any applicative structure based on (an extension of) the standard type frame, define $F_{\rho,\sigma} : (D^\rho \to D^\sigma) \to (D^{F[\rho]} \to D^{F[\sigma]})$ for each closed structure functor F and closed ρ and σ by

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$t_{\rho,\sigma}\ g\ x = g\ x \qquad (F_0 + F_1)_{\rho,\sigma}\ g\ x = \underline{\mathrm{Case}}(x, \lambda\!\!\lambda y.\underline{\mathrm{Inj}}^0((F_0)_{\rho,\sigma}\ g\ y), \lambda\!\!\lambda y.\underline{\mathrm{Inj}}^1((F_1)_{\rho,\sigma}\ g\ y))$$
$$(\sigma_0)_{\rho,\sigma}\ g\ x = x \qquad (F_0 \times F_1)_{\rho,\sigma}\ g\ x = \underline{\mathrm{Pair}}((F_0)_{\rho,\sigma}\ g\ (\underline{\mathrm{Proj}}^0 x), (F_1)_{\rho,\sigma}\ g\ (\underline{\mathrm{Proj}}^1 x)),$$
$$(\sigma_0 \to F)_{\rho,\sigma}\ g\ x = \lambda\!\!\lambda y.F_{\rho,\sigma}\ g\ (x\ y)$$

Lemma 7. *If* $\mathbf{U}$ *is an applicative structure based on an extension of the standard type frame that is a model for the fragment of the recurrence language that omits* fold_δ*,* $\Gamma, y : \rho \vdash e' : \sigma$*,* $\Gamma \vdash e : F[\rho]$*, and* η *is a* Γ*-environment, then*

$$[\![F[(\rho : y).e', e]]\!]\eta = ([\![F]\!]\eta)_{[\![\rho]\!]\eta,[\![\sigma]\!]\eta}\ (\lambda\!\!\lambda a.[\![e']\!]\eta\{y \mapsto a\})\ ([\![e]\!]\eta).$$

Proof By induction on F. ■

Combining Proposition 3 with Lemma 7, we conclude that to define a model of the recurrence language, it suffices to define an extension of the standard type frame and the following applicative structure data:

- The sets D^τ, along with an argument that D^τ is a complete upper semi-lattice;
- The semantic functions for arrow, product, and sum types;
- $\underline{\mathrm{C}}_F$ for each structure functor F.

From this data, we can define $\underline{\mathrm{D}}_F(x) = \bigvee\{z \mid \underline{\mathrm{C}}_F \le x\}$, $F_{\rho,\sigma}$ as just given, and $\underline{\mathrm{Fold}}_{F,\sigma}$ by (std-fold). Of course, there are models that are not constructed this way; Section 7.5 gives an example that is useful for extracting recurrences for lower bounds.

6.3 Syntactic sugar

We now introduce some syntactic sugar that will make our discussion of recurrences somewhat more pleasant. To simplify the discussion, we restrict the details to the source language type σ `tree` and its recurrence language potential σ tree, but we will use analogous notation for other datatypes such as `nat` and α `list` in our examples. Many of our source-language functions are really structural folds over some standard datatype—that is, the step function is a `case` expression where the argument for each branch is really the argument to one of the datatype constructors. Accordingly, we introduce notation for such `fold` expressions: for $y \notin \mathrm{fv}(e_{\mathtt{emp}}) \cup \mathrm{fv}(e_{\mathtt{node}})$,

$$\mathtt{fold}_{\sigma\ \mathtt{tree}}\ e\ \mathtt{of}\ \mathtt{emp} \Rightarrow e_{\mathtt{emp}} \mid \mathtt{node} \Rightarrow (x, r_0, r_1).e_{\mathtt{node}}$$

is syntactic sugar for

$$\mathtt{fold}_{\sigma\ \mathtt{tree}}\ e\ \mathtt{of}\ w.\mathtt{case}\ w\ \mathtt{of}\ y.e_{\mathtt{emp}}; y.e_{\mathtt{node}}\{\pi_0\ y, \pi_1\ y, \pi_2\ y/x, r_0, r_1\}.$$

We introduce a similar notation in the recurrence language:

$$\mathsf{fold}_{\sigma\ \mathsf{tree}}\ e\ \mathsf{of}\ \{\mathsf{emp} \Rightarrow e_{\mathsf{emp}} \mid \mathsf{node} \Rightarrow (x, r_0, r_1).e_{\mathsf{node}}\}$$

is syntactic sugar for

$$\mathsf{fold}_{\sigma\ \mathsf{tree}}\ e\ \mathsf{of}\ (w : F_{\sigma\ \mathsf{tree}}[\rho]).(\mathsf{case}\ w\ \mathsf{of}\ y.e_{\mathsf{emp}}; y.e_{\mathsf{node}}\{\pi_0\ y, \pi_1\ y, \pi_2\ y/x, r_0, r_1\})$$

where w and y are fresh variables.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

It would be nice to establish an identity of the form $\|\mathtt{fold}_{\sigma\ \mathtt{tree}} \cdots\| = \mathsf{fold}_{\sigma\ \mathsf{tree}} \cdots$, but the size-order axioms, which give us only inequalities, are too weak. However, the models that we will consider validate many equations, so we can set out a nice relationship. In the following proposition, we say "in the semantics, $e = e'$" to mean that for any η, $[\![e]\!]\eta = [\![e']\!]\eta$:

Proposition 5. *Suppose that we have a model such that*

- *In the semantics: if* $\|e'\|_c = 0$*, then* $\|e\{e'/x\}\| = \|e\|\{\|e'\|_p/x\}$*; and*
- *In the semantics:* $c +_c \mathsf{case}\ e\ \mathsf{of}\ \{x.e_i\}_{i=0,1} = \mathsf{case}\ e\ \mathsf{of}\ \{x.c +_c e_i\}_{i=0,1}$.

If $\Gamma \vdash \mathtt{fold}_{\sigma\ \mathtt{tree}}\ e\ \mathtt{of}\ \mathtt{emp} \Rightarrow e_{\mathtt{emp}} \mid \mathtt{node} \Rightarrow (x, r_0, r_1).e_{\mathtt{node}} : \rho$*, then in the semantics,*

$$\|\mathtt{fold}_{\sigma\ \mathtt{tree}}\ e\ \mathtt{of}\ \mathtt{emp} \Rightarrow e_{\mathtt{emp}} \mid \mathtt{node} \Rightarrow (x, r_0, r_1).e_{\mathtt{node}}\| = c +_c \mathsf{fold}_{\sigma\ \mathsf{tree}}\ p\ \mathsf{of}\ \{\mathsf{emp} \Rightarrow 1 +_c \|E_{\mathsf{emp}}\| \mid \mathsf{node} \Rightarrow (x, r_0, r_1).1 +_c \|E_{\mathsf{node}}\|\}$$

where $(c, p) = \|e\|$.

While the models that we discuss in subsequent sections satisfy the hypotheses of Proposition 5, they are not necessarily satisfied in an arbitrary model. That requires additional axioms that correspond roughly to η axioms.

7 Examples

7.1 The standard model

For the standard model, we first extend the standard type frame by including the constant $\bot$ in $\mathbf{U}_{sm}$. A semantic type (scheme) is *proper* if it has no occurrences of $\bot$. The proper semantic types (type schemes) correspond exactly to the closed syntactic types (type schemes). In the definitions of $\underline{\mu}$ and $\underline{\forall}$, we take F and τ to be proper. We define the sets A^σ by induction on σ as follows:[5]

- $A^{\mathsf{C}} = \mathbf{N}$, the natural numbers.
- $A^{\bot} = \emptyset$.
- $A^{\mathsf{unit}} = \{*\}$, some one-element set.
- $A^{\sigma_0 \to \sigma_1} = (A^{\sigma_1})^{A^{\sigma_0}}$, the set of functions from A^{σ_0} to A^{σ_1}.
- $A^{\sigma_0 \times \sigma_1} = A^{\sigma_0} \times A^{\sigma_1}$, where $\times$ is the standard set-theoretic product.
- $A^{\sigma_0 + \sigma_1} = A^{\sigma_0} \sqcup A^{\sigma_1}$, where $\sqcup$ is the standard set-theoretic disjoint union.
- $A^{\mu t.F} = \bigcup_i A^{(\lambda V.[\![F]\!]\{t \mapsto V\})^i \bot}$.
- $A^{\forall \alpha.\tau} = \prod_{\sigma \in \mathbf{U}_{sm}} A^{\tau\{\sigma/\alpha\}}$.

Define $a \leq_{A^\sigma} b$ iff $a = b$, and let the semantic functions for arrows, products, and sums be the identity functions. The definitions of $\underline{\mathrm{C}}_F$, $\underline{\mathrm{D}}_F$, and $\underline{\mathrm{Fold}}_{F,\sigma}$ are based on the standard

[5] The collection of sets $\{A^\sigma\}_{\sigma \in \mathbf{U}_{sm}}$ must be contained in some set that contains $\emptyset$ and a one-element set and is closed under disjoint unions, products, function spaces, unions of chains, and products indexed by $\mathbf{U}_{sm}$. V^{ω_1} in the standard set-theoretic hierarchy suffices.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

initial-algebra semantics. Note that we cannot use (std-fold) because the A^σ are not complete upper semi-lattices, and hence the hypotheses of Proposition 3 do not hold, and hence ($\beta_{\delta\mathsf{fold}}$) must be verified directly.

At first blush, this model is not particularly interesting. There is no abstraction of values to sizes and the "order" on costs is the identity, so the recurrences extracted from source language programs describe the exact cost of those programs in terms of the argument values. However, this is a standard model of (predicative) polymorphism, and so we can hope that parametricity may have some interesting consequences. Free theorems (Wadler, 1989) have been used to obtain relative cost information, and we discuss this further in Section 9. Here, we apply parametricity to the recurrence language and sketch the argument that if $g : \alpha\ \mathtt{list} \to \alpha\ \mathtt{list}$, then the cost of $g(xs)$ depends only on the length of xs (the same can be said for the length of $g(xs)$, but this follows from parametricity applied to the source language). For any ρ, let us define $T_\rho(xs) = ((\llbracket \|g\| \rrbracket\ \langle\!\langle \rho \rangle\!\rangle)_p(xs))_c$, the exact cost of evaluating $g(xs)$ (since $\|\cdot\|$ is a monadic translation and the interpretation of inductive types is the standard one, syntactic values of list type in the source language are isomorphic to the semantic values in the model). The goal is to show that if $xs : \rho\ \mathtt{list}$ and $ys : \sigma\ \mathtt{list}$ are of the same length, then $T_\rho(xs) = T_\sigma(ys)$. To do so, we apply parametricity to $\lambda\!\!\lambda\rho.\lambda\!\!\lambda xs.T_\rho(xs) \in A^{\forall\rho.\rho\,\mathsf{list}\to\mathsf{C}}$. We take the relational interpretation of C to be equality (so the cost constants 0 and + preserve the relation). Expanding the definition of parametricity, this means that for any ρ and σ and relation $R \subseteq A^{\langle\!\langle\rho\rangle\!\rangle} \times A^{\langle\!\langle\sigma\rangle\!\rangle}$, for any $xs \in A^{\langle\!\langle\rho\rangle\!\rangle\,\mathsf{list}}$ and $ys \in A^{\langle\!\langle\sigma\rangle\!\rangle\,\mathsf{list}}$, if $R\,\mathsf{list} \subseteq A^{\langle\!\langle\rho\rangle\!\rangle\,\mathsf{list}} \times A^{\langle\!\langle\sigma\rangle\!\rangle\,\mathsf{list}}$ holds for xs and ys, then the relational interpretation of C holds for $T_\rho(xs)$ and $T_\sigma(ys)$. Since the relational interpretation of the cost type is equality, this would give the result, so it suffices to show that there is an R such that $(R\,\mathsf{list})(xs, ys)$ holds whenever xs and ys have the same length. However, the standard relational lifting $R\,\mathsf{list}$ holds whenever xs and ys have the same length and xs_i is related to ys_i by R, so taking R to be the total relation achieves this. We conclude that if xs and ys have the same length, then the cost of $g(xs)$ and $g(ys)$ is the same.

7.2 Constructor size and height

We now describe a model in which a value v of inductive type δ is interpreted either by the number of δ constructors in v (constructor size) or by the maximum nesting depth of δ-constructors in v (constructor height), so that it reflects common size abstractions such as list length, tree size, and tree height. For example, in this model, the interpretation of the recurrence extracted from a function with domain $\sigma\ \mathtt{list}$ describes the cost in terms of the length of the argument list. For concreteness we will define the constructor size model. For the interpretation of the types, we will need two versions of the natural numbers: $\mathbf{N}_0^\infty = \{0, 1, \ldots, \infty\}$ for costs, and $\mathbf{N}_1^\infty = \{1, 2, \ldots, \infty\}$ for sizes of inductive values, which must be at least 1 because every value contains at least one constructor. $\mathbf{N}_i^\infty$ is ordered by $x \leq_{\mathbf{N}_i^\infty} y$ if $y = \infty$ or $x \leq_{\mathbf{N}} y$. The presence of ∞ may be perplexing, since all programs in the source language terminate. However, it is not always possible to give a finite upper bound on cost or potential in terms of the potential of the argument, because the notion of potential used in this model may not identify all possible sources of recursive calls. For example, consider the function `sumtree` defined in Figure 22 that sums the nodes of a `nat tree`. The cost and size of `sumtree` t depend on the size of t and the sizes of its labels, whereas

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

in this model, the potential of t only tells us the former. Since `sumtree` is definable in our source language, its recurrence can be extracted, and hence must have a meaning in this model; the only sensible interpretation is one that maps every tree size to the trivial upper bound of ∞ for both cost and potential.

We start by extending the standard type frame with additional small types $\mathbf{N}_0, \mathbf{N}_1 \in \mathbf{U}_{sm}$. Then we define the sets V^τ, observing that each V^τ is a complete upper semi-lattice. This allows us to construct a model by just defining $\underline{\mathrm{C}}_F$. The sets V^τ are defined as follows:

- $V^{\mathbf{N}_i} = \mathbf{N}_i^\infty$.
- $V^{\mathsf{C}} = \mathbf{N}_0^\infty$ with the standard interpretations for $\underline{0}$ and $\underline{+}$, where $x \underline{+} \infty = \infty \underline{+} x = \infty$.
- $V^{\mathsf{unit}} = \{*\}$.
- $V^{\sigma_0 \to \sigma_1}$ = the set of monotone functions from V^{σ_0} to V^{σ_1} with the usual pointwise order, taking $\underline{\mathrm{Abs}}$ and $\underline{\mathrm{App}}$ to be the identity functions.
- $V^{\sigma_0 \times \sigma_1} = V^{\sigma_0} \times V^{\sigma_1}$ with the usual component-wise order, taking $\underline{\mathrm{Pair}}$ and $\underline{\mathrm{Proj}}$ to be the standard pairing and projection functions.
- $V^{\sigma_0 + \sigma_1} = \mathscr{O}(V^{\sigma_0} \sqcup V^{\sigma_1})$, which we define in Section 7.2.1.
- $V^{\mu t.F} = \mathbf{N}_1^\infty$. We define $\underline{\mathrm{C}}_F$ in Section 7.2.2 (recall that we write $\underline{\mathrm{C}}_F$ for $\underline{\mathrm{C}}_{\lambda V.[\![F]\!]\{V/t\}}$, etc., and that we can define $\underline{\mathrm{D}}_F$ and $\underline{\mathrm{Fold}}_{F,\sigma}$ from it).
- $V^{\forall \alpha.\tau} = \prod_{\sigma \in \mathbf{U}_{sm}} V^{\tau\{\sigma/\alpha\}}$, with the pointwise order, taking $\underline{\mathrm{TyAbs}}$ and $\underline{\mathrm{TyApp}}$ to be the identity functions.

Once we define the interpretation of sums and datatypes, it is straightforward to verify that this is a model.

Proposition 6. $\mathbf{V} = \{V^\tau\}_{\tau \in \mathbf{U}_{lg}}$ *is a model of the recurrence language.*

Proof Since $\underline{\mathrm{Abs}}$ and $\underline{\mathrm{TyAbs}}$ are total, it suffices to verify the conditions on the semantic functions. This is trivial for arrows, products, and type quantification; sums and inductive types are handled in the next two sections. ■

7.2.1 Interpretation of sums

As we observed, we need to ensure that all the sets V^σ are complete upper semi-lattices. Preserving the complete upper semi-lattice property is straightforward for all type constructors except sum. We could take the usual disjoint sum along with a new infinite element ∞ that is a common upper bound of elements on both sides, but that ends up leading to very weak bounds in practice. For example, recall that $\underline{\mathrm{D}}_{\sigma\ \mathsf{list}}(2)$ should tell us about the data that can be used to construct a list of size ≤ 2 (which is a cons list, because we count the number of $\mathsf{c}_{\sigma\ \mathsf{list}}$ constructors, so nil has size 1). If we were to interpret sums as just proposed, both $\underline{\mathrm{Inj}}^0(*)$ and $\underline{\mathrm{Inj}}^1(a, 0)$ are such values, and their least upper bound would be ∞. It is not hard to parlay this into an argument that if $\mathtt{tail} = \lambda xs.\mathtt{case}\ xs\ \mathtt{of}\ x.\mathtt{nil}; x.\pi_1\ x$ is the usual tail function on σ `list`, then the recurrence extracted from `tail` gives a bound of ∞ for all lists of length > 1. While correct, this is hardly satisfying!

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

Instead, we take inspiration from abstract interpretation (Cousot & Cousot, 1977): we will define $\underline{\mathrm{D}}_F(n)$ to be the *set* of values x such that $\underline{\mathrm{C}}_F(x) \leq n$. We can arrange this for the typical cases of interest (i.e., finitary inductive datatypes such as lists and trees) by defining $V^{\sigma_0+\sigma_1}$ to be the downward closed subsets of $V^{\sigma_0} \sqcup V^{\sigma_1}$. We could arrange this for *all* inductive datatypes if we were to do something similar in the interpretation of arrows and products, but that entails some additional notational cost in reasoning about extracted recurrences while providing no benefits for the examples that we present. We start with some standard order-theoretic and set-theoretic definitions:

- For any partially ordered set A, the *order ideal* of A is
$$\mathscr{O}(A) =_{df} \{X \subseteq A \mid x \in X \text{ and } y \leq x \Rightarrow y \in X\}.$$
$\mathscr{O}(A)$ is partially ordered by set inclusion and is a complete upper semi-lattice; concretely, if $X \subseteq \mathscr{O}(A)$, then $\bigvee X = \bigcup X$.
- For any $X \subseteq A$, $\downarrow^A X = \{x \in A \mid \exists y \in X. x \leq y\} \in \mathscr{O}(A)$ and for $a \in A$, $\downarrow^A a = \downarrow\{a\}$ (we drop the superscript when it is clear from context).
- For any $f : A \to B$ and $X \subseteq A$, $f[X] = \{f(x) \mid x \in X\}$.
- If X_0 and X_1 are partially ordered sets, $X_0 \sqcup X_1$ is the usual disjoint union with injection functions $\mathrm{in}^i : X_i \to X_0 \sqcup X_1$ partially ordered by $x \leq y$ iff $x = \mathrm{in}^i(x')$, $y = \mathrm{in}^i(y')$, and $x' \leq_{X_i} y'$.

For the interpretation of sums, we define $V^{\sigma_0+\sigma_1} = \mathscr{O}(V^{\sigma_0} \sqcup V^{\sigma_1})$ with the semantic functions defined by

$$\underline{\mathrm{Inj}}^i(x) = \mathrm{in}^i[\downarrow^{V^{\sigma_i}} x] = \downarrow^{V^{\sigma_0} \sqcup V^{\sigma_1}} (\mathrm{in}^i(x))$$
$$\underline{\mathrm{Case}}(X_0 \sqcup X_1, f_0, f_1) = \bigvee f_0[X_0] \vee \bigvee f_1[X_1]$$

Lemma 8. $\underline{\mathrm{Case}}(\underline{\mathrm{Inj}}^i(x), f_0, f_1) \geq f_i(x)$.

Proof

$$\begin{aligned}
\underline{\mathrm{Case}}(\underline{\mathrm{Inj}}^i(x), f_0, f_1) &= \underline{\mathrm{Case}}(\mathrm{in}^i[\downarrow x] \sqcup \emptyset, f_0, f_1) \\
&= \bigvee f_i[\downarrow x] \vee \bigvee f_{1-i}[\emptyset] \\
&= \bigvee f_i[\downarrow x] \\
&\geq f_i(x) \qquad (x \in \downarrow x).
\end{aligned}$$

■

Note that $\mathscr{O}A$ is a monad on the category of partially ordered sets and monotone functions, with unit $A \to \mathscr{O}A$ given by $\downarrow^A$, and multiplication $\mathscr{O}\mathscr{O}A \to \mathscr{O}A$ given by union, and it plays the role of a powerset monad on posets (the ordinary powerset operation, without the additional downward closure requirement, does not have a *monotone* function $A \to \mathscr{P}(A)$, because $x \leq_A y$ does not imply that $\{x\} \subseteq \{y\}$. When a partially ordered set is a complete upper semilattice (i.e. supports the maximum operation that we use to interpret the recursor), it is an *algebra* for this monad, i.e. there is a monotone function $\mathscr{O}A \to A$, satisfying some equations. Thus, another way of understanding these models

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

is that, for functions and products, we build algebras $\bigvee_{A\to B} : \mathscr{O}(A \to B) \to (A \to B)$ and $\bigvee_{A\times B} : \mathscr{O}(A \times B) \to A \times B$ from algebra structures $\bigvee_A : \mathscr{O}A \to A$ and $\bigvee_B : \mathscr{O}B \to B$, but for sums, we use the free algebra $\mathscr{O}(A + B)$, with $\bigvee_{\mathscr{O}(A+B)} : \mathscr{O}\mathscr{O}(A + B) \to \mathscr{O}(A + B)$ given by union.

7.2.2 Semantic functions for inductive datatypes

We define $\underline{\mathrm{C}}_F$ by first defining a function $\underline{\mathrm{size}}_F : V^{F[\mu t.F]} \to \mathbf{N}_0^\infty$. For $\delta = \mu t.F$, a semantic value of type $F[\mu t.F]$ represents the data from which a value of type δ is constructed, but with the inductive substructures replaced by their sizes, and $\underline{\mathrm{size}}_F$ returns the size of the inductive value constructed from that data.

$$\underline{\mathrm{size}}_t(n) = n \qquad \underline{\mathrm{size}}_{F_0+F_1}(X_0 \sqcup X_1) = \bigvee \underline{\mathrm{size}}_{F_0}[X_0] \vee \bigvee \underline{\mathrm{size}}_{F_1}[X_1]$$
$$\underline{\mathrm{size}}_\sigma(x) = 0 \qquad \underline{\mathrm{size}}_{F_0\times F_1}(a_0, a_1) = \underline{\mathrm{size}}_{F_0}(a_0) + \underline{\mathrm{size}}_{F_1}(a_1)$$
$$\underline{\mathrm{size}}_{\sigma\to F}(g) = \sum\{\underline{\mathrm{size}}_F(g\,x) \mid x \in V^\sigma\}$$

For the constructor height model, define a function $\underline{\mathrm{height}}_F$ analogously, replacing the sums in the product and arrow shapes with maximums. The semantic constructor and destructor are then defined by

$$\underline{\mathrm{C}}_F(a) = 1 + \underline{\mathrm{size}}_F(a) \qquad \underline{\mathrm{D}}_F(n) = \bigvee\{a \mid \underline{\mathrm{C}}_F\, a \le n\}.$$

To use (std-fold), it suffices to verify the conditions of Proposition 3, which is trivial, so we have

$$\underline{\mathrm{Fold}}_{F,\sigma}\, s\, x = \bigvee\{s\big(F_{\delta,\sigma}(\underline{\mathrm{Fold}}_{F,\sigma}\, s)z\big) \mid 1 + \underline{\mathrm{size}}_F(z) \le x\}.$$

7.2.3 Examples: lists and trees

Referring to Figure 10,

$$[\![\_ \vdash \mathsf{nil} : \sigma\ \mathsf{list}]\!]\eta = 1$$
$$[\![x : \sigma, xs : \sigma\ \mathsf{list} \vdash \mathsf{cons}(x, xs) : \sigma\ \mathsf{list}]\!]\eta = 1 + \eta(xs)$$
$$[\![\_ \vdash \mathsf{emp} : \sigma\ \mathsf{tree}]\!]\eta = 1$$
$$[\![x : \sigma, t_0 : \sigma\ \mathsf{tree}, t_1 : \sigma\ \mathsf{tree} \vdash \mathsf{node}(x, t_0, t_1) : \sigma\ \mathsf{tree}]\!]\eta = 1 + \eta(t_0) + \eta(t_1).$$

It is not hard to see that $[\![\_ \vdash t : \sigma\ \mathsf{tree}]\!] = 2n + 1$, where n is the usual size of t (i.e., $[\![\_ \vdash t : \sigma\ \mathsf{tree}]\!]$ is the number of internal and external nodes of t). Since this is linear in the usual notion of size of a tree, it suffices for showing that the recurrences that we extract have the expected O-behavior.

Destructors exhibit the desired behavior; consider σ list again:

$$\underline{\mathrm{D}}_{F_\sigma\,\mathsf{list}}(x) = \begin{cases} \{*\} \sqcup \emptyset, & x = 1 \\ \{*\} \sqcup \{(a, x') \mid a \in V^\sigma, 1 + x' \le x\}, & 2 \le x \le \infty \end{cases}$$
$$= \{*\} \sqcup (V^\sigma \times \downarrow^{\mathbf{N}_1^\infty}(x - 1))$$

where we define $\downarrow^{\mathbf{N}_1^\infty} 0 = \emptyset$. In other words, a list of size 1 must be `nil` and a list of length at most x is either `nil` or `cons`(x, xs), where xs has length at most $x - 1$. Remember that $V^{F[\sigma\ \mathsf{list}]} = \mathscr{O}(\{*\} \sqcup (V^\sigma \times \mathbf{N}_1^\infty))$, so if $\mathrm{in}^1(a, x) \in X \in V^{F[\sigma\ \mathsf{list}]}$, then $x \ge 1$; that is why

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$\underline{\mathrm{D}}_{F_\sigma\ \mathsf{list}}(1) \neq \{*\} \sqcup X$ with $X \neq \emptyset$. For σ tree, the result is equally pleasant:

$$\underline{\mathrm{D}}_{F_\sigma\ \mathsf{tree}}(x) = \begin{cases} \{*\} \sqcup \emptyset, & x = 1 \\ \{*\} \sqcup \{(a, x_0, x_1) \mid a \in V^\sigma, 1 + x_0 + x_1 \leq x\}, & 2 \leq x \leq \infty \end{cases}$$

Finally, we observe the following simple forms for the denotation of recurrences over lists and trees:

Proposition 7.

1. *If* $f\ n = [\![\mathsf{fold}_{\sigma\ \mathsf{list}}\ y\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons} \Rightarrow (x, r).e_{\mathsf{cons}}\}]\!]\eta\{y \mapsto n\}$, *then in the constructor size and height models,*

$$\begin{aligned} f\ 1 &= [\![e_{\mathsf{nil}}]\!]\eta \\ f\ n &= [\![e_{\mathsf{nil}}]\!]\eta \vee \bigvee\{[\![e_{\mathsf{cons}}]\!]\eta\{x, r \mapsto \infty^\sigma, f\ n'\} \mid n' < n\} \\ &= [\![e_{\mathsf{nil}}]\!]\eta \vee [\![e_{\mathsf{cons}}]\!]\eta\{x, r \mapsto \infty^\sigma, f(n-1)\} \qquad (n > 1). \end{aligned}$$

   The second form for $f\ n$, $n > 1$, *follows from monotonicity of the denotation function.*

2. *If* $f\ n = [\![\mathsf{fold}_{\sigma\ \mathsf{tree}}\ y\ \mathsf{of}\ \{\mathsf{emp} \Rightarrow e_{\mathsf{emp}} \mid \mathsf{node} \Rightarrow (x, r_0, r_1).e_{\mathsf{node}}\}]\!]\eta\{y \mapsto n\}$, *then in the constructor size model,*

$$\begin{aligned} f\ 1 &= [\![e_{\mathsf{emp}}]\!]\eta \\ f\ n &= [\![e_{\mathsf{emp}}]\!]\eta \vee \bigvee\{[\![e_{\mathsf{node}}]\!]\eta\{x, r_0, r_1 \mapsto \infty^\sigma, f\ n_0, f\ n_1\} \mid n_0 + n_1 < n\} \quad (n > 1). \end{aligned}$$

   In the constructor height model, replace $n_0 + n_1 < n$ *with* $n_0 \vee n_1 < n$.

Proof The verification is a moderately tedious calculation; here, it is for (2) with $n > 1$. Let

$$\begin{aligned} s &= \lambda\!\!\lambda(Z \sqcup X).[\![\mathsf{case}\ w\ \mathsf{of}\ y.e_{\mathsf{emp}}; y.e_{\mathsf{node}}\{\pi_0\ y, \pi_1\ y, \pi_2\ y/x, r_0, r_1\}]\!]\eta\{w \mapsto Z \sqcup X\} \\ &= \lambda\!\!\lambda(Z \sqcup X).[\![e_{\mathsf{emp}}]\!]\eta \vee \bigvee\left\{[\![e_{\mathsf{node}}]\!]\eta\{a, b_0, b_1 \mapsto x, r_0, r_1\} \mid (a, b_0, b_1) \in X\right\} \end{aligned}$$

Observe that $f = \underline{\mathrm{Fold}}\ s$ and let us write $\underline{\mathrm{Map}}$ for $([\![F_{\sigma\ \mathsf{tree}}]\!]\eta)_{[\![\sigma\ \mathsf{tree}]\!]\eta, [\![\rho]\!]\eta}$. By definition, we have $f\ n = \bigvee\{s(\underline{\mathrm{Map}}\ f\ (Z \sqcup X)) \mid \underline{\mathrm{C}}(Z \sqcup X) \leq n\}$. By monotonicity, we need only consider $Z = \{*\}$, and by definition of $\underline{\mathrm{C}}$, we need only consider nonempty sets X such that $(\_, n_0, n_1) \in X$ implies $n_0 + n_1 < n$, so

$$\begin{aligned} f\ n &= \bigvee\{s(\underline{\mathrm{Map}}\ f\ (\{*\} \sqcup X)) \mid (\_, n_0, n_1) \in X \Rightarrow n_0 + n_1 < n\} \\ &= \bigvee\{s(\{*\} \sqcup \underline{\mathrm{Map}}\ f\ (\mathrm{in}^1[X])) \mid (\_, n_0, n_1) \in X \Rightarrow n_0 + n_1 < n\} \\ &= \bigvee \Big\{ [\![e_{\mathsf{emp}}]\!]\eta \vee \bigvee\left\{[\![e_{\mathsf{node}}]\!]\eta\{x, r_0, r_1 \mapsto b, k_0, k_1\} \mid (b, k_0, k_1) \in \underline{\mathrm{Map}}\ f\ (\mathrm{in}^1[X])\right\} \\ &\qquad \mid (\_, n_0, n_1) \in X \Rightarrow n_0 + n_1 < n \Big\} \\ &= [\![e_{\mathsf{emp}}]\!]\eta \vee \bigvee \Big\{ \bigvee\left\{[\![e_{\mathsf{node}}]\!]\eta\{x, r_0, r_1 \mapsto b, k_0, k_1\} \mid (b, k_0, k_1) \in \underline{\mathrm{Map}}\ f\ (\mathrm{in}^1[X])\right\} \\ &\qquad \mid (\_, n_0, n_1) \in X \Rightarrow n_0 + n_1 < n \Big\} \end{aligned}$$

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$= [\![e_{\mathsf{emp}}]\!]\eta \vee \bigvee \Big\{ \bigvee \{ [\![e_{\mathsf{node}}]\!]\eta\{x, r_0, r_1 \mapsto b, k_0, k_1\} \mid (b, k_0, k_1) \in \bigvee \{\downarrow(a, f\ n_0, f\ n_1) \mid (a, n_0, n_1) \in X\} \} \mid (\_, n_0, n_1) \in X \Rightarrow n_0 + n_1 < n \Big\}$$

$$= [\![e_{\mathsf{emp}}]\!]\eta \vee \bigvee \Big\{ \bigvee \{ [\![e_{\mathsf{node}}]\!]\eta\{x, r_0, r_1 \mapsto b, k_0, k_1\} \mid \exists (a, n_0, n_1) \in X : (b, k_0, k_1) \leq (a, f\ n_0, f\ n_1) \} \mid (\_, n_0, n_1) \in X \Rightarrow n_0 + n_1 < n \Big\}$$

$$= [\![e_{\mathsf{emp}}]\!]\eta \vee \bigvee \{ [\![e_{\mathsf{node}}]\!]\eta\{x, r_0, r_1 \mapsto \infty^\sigma, f\ n_0, f\ n_1\} \mid n_0 + n_1 < n \}$$

Let us write the last equation as

$$L = [\![e_{\mathsf{emp}}]\!]\eta \vee \bigvee \Big\{ \bigvee A_X \mid (\_, n_0, n_1) \in X \Rightarrow n_0 + n_1 < n \Big\} = [\![e_{\mathsf{emp}}]\!]\eta \vee \bigvee B = R$$

First, let us show that for any X such that $(\_, n_0, n_1) \in X \Rightarrow n_0 + n_1 < n$, $A_X \subseteq \downarrow B$, from which we conclude that $\bigvee A_X \leq \bigvee B$, and hence $L \leq R$. For any such X, take (b, k_0, k_1) such that there is $(a, n_0, n_1) \in X$ with $(b, k_0, k_1) \leq (a, f\ n_0, f\ n_1) \leq (\infty, f\ n_0, f\ n_1)$. By Proposition 6, $[\![e_{\mathsf{node}}]\!]\eta\{x, r_0, r_1 \mapsto b, k_0, k_1\} \leq [\![e_{\mathsf{node}}]\!]\eta\{x, r_0, r_1 \mapsto \infty, f\ n_0, f\ n_1\}$. Since (b, k_0, k_1) was chosen arbitrarily, $A_X \subseteq \downarrow B$, as needed. To show that $R \leq L$, suppose that $n_0 + n_1 < n$. Then $[\![e_{\mathsf{node}}]\!]\eta\{x, r_0, r_1 \mapsto \infty, f\ n_0, f\ n_1\} \in A_{\downarrow(\infty, n_0, n_1)}$, from which $R \leq L$ follows. ■

Although we will primarily use Proposition 7, it may be instructive to work through an example of explicitly constructing $\underline{\mathrm{Fold}}_{F,\rho}$ from the proof of Proposition 3. Consider $F = \mathsf{unit} + t$ (the structure functor for `nat`) and set $s(x) = \underline{\mathrm{Case}}(x, \lambda\!\!\lambda u.1, \lambda\!\!\lambda u.1 + x)$. s might be the step function for the recurrence that describes the cost of the copy function on `nat`. Define Q as in the proof of Lemma 3. In this setting, the bottom element at which we start iterating Q is the function that is constantly 0. Set $f_0 = Q \perp$ and $f_{n+1} = Q f_n$. Just as in the calculation of $\underline{\mathrm{D}}_{F_\sigma\ \mathsf{list}}$,

$$\underline{\mathrm{D}}_{F_{\mathsf{nat}}}(x) = \begin{cases} \{*\} \sqcup \emptyset, & x = 1 \\ \{*\} \sqcup \{x' \mid 1 + x' \leq x\}, & 2 \leq x \end{cases}$$

and so a bit more calculation shows that

$$f_k(n) = \begin{cases} n, & n \leq k + 1 \\ k + 1, & n > k + 1. \end{cases}$$

It is not hard to see that $f_\omega(n) = n$ is a fixed point of Q, so we conclude that $[\![\mathsf{fold}_{\mathsf{nat}}\ x\ \mathsf{of}\ \{\mathsf{Z} \Rightarrow 1 \mid \mathsf{S} \Rightarrow r.\mathsf{S}r\}]\!]\{x \mapsto n\} = n$.

Of course, this is precisely what we expect, though for readers familiar with how a typical recursive function on numbers is defined by successive approximations, the route may feel a bit different. Usually when defining a recursive function on numbers, one takes the flat order and starts with the everywhere-undefined function. For a typical total function, the kth approximation is a partial function that is defined and correct on some initial segment of the natural numbers and undefined elsewhere. Here we take the (more-or-less) standard order and start with a function that is everywhere an unlikely bound (namely, 0).

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\mathsf{copy}_{\sigma\ \mathtt{tree}} = \lambda t.\mathtt{fold}_{\sigma\ \mathtt{tree}}\ t\ \mathtt{of}\quad \mathsf{emp} \Rightarrow \mathsf{emp}$$
$$\mid \mathsf{node} \Rightarrow (x, r_0, r_1).\mathsf{node}(x, \mathtt{force}\ r_0, \mathtt{force}\ r_1).$$

$$\mathsf{copy}_{\sigma\ \mathtt{tree}} = \lambda t.\mathsf{fold}_{\sigma\ \mathtt{tree}}\ t\ \mathsf{of}\ \left\{ \begin{array}{l} \mathsf{emp} \Rightarrow (1, 1) \\ \mid \mathsf{node} \Rightarrow (x, r_0, r_1).(1 + r_{0c} + r_{1c}, \mathsf{node}(x, r_{0p}, r_{1p})) \end{array} \right\}.$$

Fig. 20: The monomorphic tree copy function and its extracted recurrence.

Each successive approximation yields a function with more likely bounds, terminating with a (hopefully low but) correct bound. In the case of a partial recursive function, the "bad" case is that the function is not defined for some numbers (the value of the approximants never gets above $\bot$). In our setting, the "bad" case is that the bound is infinite (the value of the approximants never stops growing). The reader may wish to compare this with the use of $\mathbf{N}_0^\infty$ by Rosendahl (1989), where ∞ corresponds to the bottom element in the usual CPO semantics for fixpoints. We return to this in Section 8 when we discuss general recursion.

7.2.4 Example: tree copy

For a first "sanity check," let us analyze the tree copy function that is defined in Figure 20. We will also describe some of the main features in the analysis that are typical of all of our examples. The first is that a source language program $e = \lambda x, y, z.e'$ extracts to a recurrence of the form $(0, \lambda x.(0, \lambda y.(0, \lambda z.\|e'\|)))$. However, we are really only interested in $\|e'\|$ as a function of the potentials x, y, and z. Accordingly, when analyzing a program such as e, we focus on the recurrence language program $\lambda x, y, z.\|e'\|$. Here, this means we will analyze the (denotation of the) recurrence $\mathsf{copy}_{\sigma\ \mathtt{tree}}$ that is also shown in Figure 20. Second, we shall use Proposition 5 freely as though it is a theorem about the syntax when we write our examples. Third, in our examples, we typically use the identifier r in syntactic recurrences for a recursive call to the computation of a complexity, and hence r_p and r_c correspond to recursive calls that compute potential and cost, respectively. Finally, we remind the reader that our goal is to show that the semantic recurrences are essentially the same as those that we expect to arise from an informal analysis, and so we make no attempt to solve them.

The analysis for $\mathsf{copy}_{\sigma\ \mathtt{tree}}$ proceeds as follows. Define $T(n) = (\llbracket \mathsf{copy}_{\sigma\ \mathtt{tree}} \rrbracket(n))_c$. Following the definition of the denotation function and using Proposition 7 and facts about $\vee$ and $\bigvee$ in the semantics, we have

$$T(1) = 1 \qquad T(n) = 1 \vee \bigvee\{1 + T(n_0) + T(n_1) \mid n_0 + n_1 < n\}$$
$$= \bigvee\{1 + T(n_0) + T(n_1) \mid n_0 + n_1 < n\}$$

and we obtain a similar recurrence for $S(n) = (\llbracket \mathsf{copy}_{\sigma\ \mathtt{tree}} \rrbracket(n))_p$. We observe that these are precisely the expected recurrences from an informal analysis which, if one is careful, must consider all possible combinations of subtree sizes when computing the cost or size of the result when the argument tree has size n.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\begin{aligned}\texttt{mem} = \lambda cmp^{\sigma\times\sigma\to\texttt{order}}, t^{\sigma\ \texttt{tree}}, x^{\sigma}.\\ \texttt{fold}_{\sigma\ \texttt{tree}}\, t\ \texttt{of}\ \ \texttt{emp} \Rightarrow \texttt{false}\\ \quad \mid \texttt{node} \Rightarrow (y, r_0, r_1).\texttt{case}\ cmp\ (x, y)\ \texttt{of}\ \texttt{LT.force}\ r_0;\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{EQ.true};\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{GT.force}\ r_1\end{aligned}$$

$$\begin{aligned}&\mathsf{mem} =\\ &\lambda(cmp : \langle\!\langle\sigma\rangle\!\rangle \times \langle\!\langle\sigma\rangle\!\rangle \to \|\mathsf{order}\|), (h : \langle\!\langle\sigma\rangle\!\rangle\ \mathsf{tree}), (x : \langle\!\langle\sigma\rangle\!\rangle).\\ &\mathsf{fold}_{\langle\!\langle\sigma\rangle\!\rangle\ \mathsf{tree}}\, h\ \mathsf{of} \left\{\begin{array}{l} \mathsf{emp} \Rightarrow (1, \mathsf{false})\\ \mid \mathsf{node} \Rightarrow (y, r_0, r_1).\mathsf{case}\ (cmp\ (x, y))_p\ \mathsf{of}\ \mathsf{LT}.(1 + (cmp\ (x, y))_c + r_{0c}, r_{0p});\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{EQ}.(1, \mathsf{true});\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{GT}.(1 + (cmp\ (x, y))_c + r_{1c}, r_{1p}) \end{array}\right\}\end{aligned}$$

Fig. 21: Binary search tree membership and its extracted recurrence.

7.2.5 Example: binary search tree membership

For an interesting example, let us consider membership testing in σ-labeled binary search trees. First, we define the type

$$\texttt{order} = \texttt{unit} + \texttt{unit} + \texttt{unit}$$

and write $\texttt{case}\ e\ \texttt{of}\ \texttt{LT}.e_0; \texttt{EQ}.e_1; \texttt{GT}.e_2$ for $\texttt{case}^{\texttt{order}}\ e\ \texttt{of}\ x.e_0; x.e_1; x.e_2$, and we assume comparable notation in the recurrence language. The membership test function is given in Figure 21.

Let us consider an informal analysis of `mem`, which is somewhat simpler to describe in reference to the `member` function of Figure 8(a). Let $T(h)$ be the number of calls to `member` in terms of the height of t. We would probably argue that $T(1) = 1$ and for $h > 1$,

$$T(h) \leq \underbrace{1}_{\text{the call to member}} + \underbrace{\bigvee\{0, T(h_0), T(h_1)\}}_{\text{cost of a case is bounded by the costs of its branches}},$$

where $h_0, h_1 < h$. But since the only information we have is that t_0 and t_1 are subtrees of some tree t of height h, what we must really mean is that

$$T(h) \leq 1 + \bigvee\{0, T(h_0), T(h_1) \mid h_0 \vee h_1 < h\},$$

so this is the recurrence we expect to see in a formal analysis.

Taking the same approach as in the previous section, we analyze the recurrence mem given in Figure 21, this time considering its denotation in the constructor height model. The extracted recurrence makes explicit the dependence of the complexity of `mem` on the complexity of the comparison function *cmp*. Of course, a typical analysis will make assumptions about this complexity. The most common such (and the one we implicitly made in our informal analysis) is that the cost of the comparison function is independent of the size of its arguments, which we can model here by assuming that $(cmp\ (x, y))_c = 0$ for all x and y (more precisely, we only analyze $[\![\mathsf{mem}]\!]\ cmp$ under the assumption that *cmp* satisfies this condition). Define $T(h) = ([\![\mathsf{mem}]\!]\ cmp\ h\ x)_c$ and assume that $cmp(x, \infty) =$

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$A \sqcup B \sqcup C$. Then making use of Proposition 7,

$$
\begin{aligned}
T(1) = 1 \qquad T(h) &= 1 \vee \bigvee \Big\{ \bigvee \{1 + (mem\ cmp\ h_0\ x)_c \mid a \in A\} \vee \\
&\qquad\qquad \bigvee \{1 \mid b \in B\} \vee \\
&\qquad\qquad \bigvee \{1 + (mem\ cmp\ h_1\ x)_c \mid c \in C\} \\
&\qquad\qquad \mid h_0 \vee h_1 < h \Big\} \\
&= 1 \vee \bigvee \Big\{ \bigvee \{1 + T(h_0) \mid a \in A\} \vee \\
&\qquad\qquad \bigvee \{1 \mid b \in B\} \vee \\
&\qquad\qquad \bigvee \{1 + T(h_1) \mid c \in C\} \\
&\qquad\qquad \mid h_0 \vee h_1 < h \Big\} \\
&\leq 1 \vee \bigvee \{1 + T(h_0), 1, 1 + T(h_1) \mid h_0 \vee h_1 < h\} \\
&= \bigvee \{1 + T(h_0), 1, 1 + T(h_1) \mid h_0 \vee h_1 < h\}.
\end{aligned}
$$

Again, we have essentially the same recurrence as given by the informal analysis. The last inequality is valid because A, B, and C are all subsets of $\{*\}$, and hence are either $\emptyset$ or $\{*\}$ itself, and we take advantage of the fact that $\bigvee \emptyset = 0$. The comparison with ∞ might be a bit perturbing. In this model, the labels do not contribute to the potential of a tree. Since the comparison in the recurrence arises from the comparison of x with an arbitrary node label y, the best we can say about the potential of y is that it is at most ∞. For another perspective, keep in mind that *cmp* is monotone, so unless it is a particularly odd function, $cmp(x, \infty) = \{*\} \sqcup \{*\} \sqcup \{*\}$, which forces the recurrence to take all possible outcomes into account. This is precisely what we would expect in an informal analysis.

7.2.6 *Inductive types as an abstract interpretation*

Our justification for the interpretation of sum types appealed to intuition from abstract interpretation. For datatypes with structure functors that are sums of products (e.g., lists and trees), the connection goes beyond just intuition, as it is easy to see that not only do we have that $\underline{\mathrm{D}} \circ \underline{\mathrm{C}} \geq \mathrm{id}$ (β_δ) but also that $\underline{\mathrm{C}} \circ \underline{\mathrm{D}} = \mathrm{id}$. This is precisely the kind of Galois connection we would expect to see in an abstract interpretation, where here we think of the datatype as being the abstract domain and its unfolding to be the concrete domain.[6] Intuitively, this is exactly how we think of models of the recurrence language as performing a size abstraction on datatypes. Interpreting a datatype value (i.e., an application of the constructor) as a size abstracts away information. Destructing a size tells us how a value of that size may be constructed from other data, but that data can only tell us the sizes of the substructures used in the construction. In other words, the application of the destructor gives us more concrete information about a size, namely, something about the composition of a value of that size.

[6] Of course, the domains here are not of finite height as in typical AI analyses, but that is typically for the benefit of computability of those analyses that would correspond to computing the denotation of the bounding recurrence, which is not our primary concern here.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\begin{aligned}
\texttt{plus} &= \lambda x, y.\texttt{fold}_{\texttt{nat}}\ x\ \texttt{of}\ \texttt{Z} \Rightarrow y \mid \texttt{S} \Rightarrow (r).\texttt{S}\ (\texttt{force}\ r)\\
\texttt{sumtree} &= \lambda t.\texttt{fold}_{\texttt{nat tree}}\ t\ \texttt{of}\quad \texttt{emp} \Rightarrow \texttt{Z}\\
&\qquad\qquad \mid \texttt{node} \Rightarrow (x, r_0, r_1).\texttt{plus}\ x\ (\texttt{plus}\ (\texttt{force}\ r_0)\ (\texttt{force}\ r_1))
\end{aligned}$$

$$\mathsf{plus} = \lambda x, y.\mathsf{fold}_{\mathsf{nat}}\ x\ \mathsf{of}\ \left\{\mathsf{Z} \Rightarrow (1, y) \mid \mathsf{S} \Rightarrow (r).(1 + r_c, \mathsf{S}\ r_p)\right\}$$

$$\mathsf{sumtree} = \lambda t.\mathsf{fold}_{\mathsf{nat\ tree}}\ t\ \mathsf{of}\ \left\{\begin{array}{l}\mathsf{emp} \Rightarrow (1, \mathsf{Z})\\ \mid \mathsf{node} \Rightarrow (x, r_0, r_1).(1 + r_{0c} + r_{1c} + (\mathsf{plus}\ r_{0p}\ r_{1p})_c) +_c\\ \qquad\qquad \mathsf{plus}\ x\ (\mathsf{plus}\ r_{0p}\ r_{1p})_p\end{array}\right\}$$

Fig. 22: A function that sums the nodes of a `nat tree`.

7.3 Counting all constructors

The cost of some functions cannot be usefully described in terms of the "usual" notion of size captured by the model **V** of the previous section. For example, to usefully analyze the `sumtree` function of Figure 22, we need a model in which the size of a nat tree value measures both the number of nat tree constructors and the number of nat constructors. In this section, we give an example of how to construct such a model. In it, a value v of inductive type is interpreted by a function ϕ such that $\phi(\delta)$ is the size of the largest maximal subtree of v that contains only δ-constructors. For $v : \mathsf{nat\ tree}$, this means that $[\![v]\!](\mathsf{nat\ tree})$ is the usual size of v, $[\![v]\!](\mathsf{nat})$ is the maximum label size of v, and $[\![v]\!](\delta) = 0$ for $\delta \notin \{\mathsf{nat\ tree}, \mathsf{nat}\}$.

Because we want to distinguish between constructors for different inductive types, it is convenient to use the following alternative grammar for types and structure functors, which just spells out the closed-type production for structure functors:

$$\begin{aligned}
\sigma &::= \alpha \mid \mathsf{C} \mid \mathsf{unit} \mid \sigma + \sigma \mid \sigma \times \sigma \mid \sigma \to \sigma \mid \mu t.F\\
F &::= t \mid \alpha \mid \mathsf{C} \mid \mathsf{unit} \mid \mu t.F \mid F + F \mid F \times F \mid \sigma \to F.
\end{aligned} \qquad (*)$$

The content of the next proposition is just that the grammar $(*)$ defines the same words as that of Figure 9.

Proposition 8.

1. *If σ is a type by the grammar $(*)$, then σ is a structure functor by the grammar $(*)$.*
2. *σ is a type by the grammar of Figure 9 iff σ is a type by the grammar $(*)$, and F is a structure functor by the grammar of Figure 9 iff F is a structure functor by the grammar $(*)$.*

Proof

1. Induction on σ.
2. Induction on the μ-nesting depth of σ and F. The main idea is that we treat t as a fixed symbol, rather than a meta-variable ranging over a class of variables, so inside the $\mu t.F$ production of F, it is no longer possible to refer to the "outer" t, and the $\mu t.F$ production of F always corresponds to a constant shape functor.

■

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

The type frame is the same as for the constructor-counting model of Section 7.2; for the current model, we write W^σ for the interpretation of σ. Except for inductive types, the clauses for W^σ are the same as those for V^σ from Section 7.2. Set $\mathscr{D} = \{\mu t.F \mid F \text{ closed}\}$ and

- $W^{\mu t.F} = \{\phi \in D \to \mathbf{N}_0^\infty \mid \phi(\mu t.F) \geq 1, \delta \text{ not a syntactic subtype of } F \Rightarrow \phi(\delta) = 0\}$.

To define $\underline{\mathrm{C}}_F$ and $\underline{\mathrm{D}}_F$, we define $\underline{\mathrm{size}}_{F,\delta} : W^{F[\delta]} \to (\mathscr{D} \to \mathbf{N}_0^\infty)$ similarly to the previous section. The additional subscript enables us to track which datatype is the "main" datatype, as the counting is different for products for the main datatype and others. The definition is as follows:

$$\underline{\mathrm{size}}_{t,\delta}(\phi) = \phi \qquad \underline{\mathrm{size}}_{F_0+F_1,\delta}(X_0 \sqcup X_1) = \bigvee \underline{\mathrm{size}}_{F_0,\delta}[X_0] \vee \bigvee \underline{\mathrm{size}}_{F_1,\delta}[X_1]$$

$$\underline{\mathrm{size}}_{\mathsf{C},\delta}(x) = \lambda\!\!\lambda\delta.0 \qquad \underline{\mathrm{size}}_{F_0\times F_1,\delta}(x_0,x_1) = \lambda\!\!\lambda\delta'. \begin{cases} \underline{\mathrm{size}}_{F_0,\delta}(x_0)(\delta') + \underline{\mathrm{size}}_{F_1,\delta}(x_1)(\delta'), & \delta' = \delta \\ \underline{\mathrm{size}}_{F_0,\delta}(x_0)(\delta') \vee \underline{\mathrm{size}}_{F_1,\delta}(x_1)(\delta'), & \delta' \neq \delta \end{cases}$$

$$\underline{\mathrm{size}}_{\mathsf{unit},\delta}(*) = \lambda\!\!\lambda\delta.0$$

$$\underline{\mathrm{size}}_{\mu t.F,\delta}(\phi) = \phi$$

$$\underline{\mathrm{size}}_{\sigma\to F,\delta}(f) = \lambda\!\!\lambda\delta'. \begin{cases} \sum\{\underline{\mathrm{size}}_{F,\delta}(f\,x)(\delta') \mid x \in D^\sigma\}, & \delta' = \delta \\ \bigvee\{\underline{\mathrm{size}}_{F,\delta}(f\,x)(\delta') \mid x \in D^\sigma\}, & \delta' \neq \delta \end{cases}$$

Set

$$\underline{\mathrm{C}}_F(a) = \lambda\!\!\lambda\delta.\chi_F(\delta) + \underline{\mathrm{size}}_{F,\mu t.F}(a)(\delta) \qquad \underline{\mathrm{D}}_F(\phi) = \bigvee\{a \mid \underline{\mathrm{C}}_F(a) \leq \phi\}$$

where $\chi_F(\delta) = 1$ if $\delta = \mu t.F$, $\chi_F(\delta) = 0$ otherwise. Notice that for $\delta = \mu t.F$, $\underline{\mathrm{C}}_F(a)(\delta) = 1 + \underline{\mathrm{size}}_{F,\delta}(a)(\delta) \geq 1$, so $\underline{\mathrm{C}}_F(a) \in W^\delta$. fold_δ is interpreted by (std-fold) as usual.

7.3.1 *Example: the potential of* nat tree

Although we could prove a general theorem to show that $\underline{\mathrm{size}}_{F,\delta}$ encapsulates the description given above, seeing the details of the specific case of nat tree is more illuminating. To start, some notation is helpful: set $\phi_n^{\mathsf{nat}} \in W^{\mathsf{nat}}$ and $\phi_{n,k}^{\mathsf{nat\,tree}} \in W^{\mathsf{nat\,tree}}$ to be the functions

$$\phi_n^{\mathsf{nat}}(\mathsf{nat}) = n \qquad \phi_{n,k}^{\mathsf{nat\,tree}}(\mathsf{nat}) = n$$
$$\phi_n^{\mathsf{nat}}(\_) = 0 \qquad \phi_{n,k}^{\mathsf{nat\,tree}}(\mathsf{nat\ tree}) = k$$
$$\phi_{n,k}^{\mathsf{nat\,tree}}(\_) = 0$$

First we start with a useful lemma:

Lemma 9. *$[\![\mathsf{S}]\!](\phi) = \chi_{F_{\mathsf{nat}}} + \phi$, and in particular, $[\![\mathsf{S}]\!](\phi_k^{\mathsf{nat}}) = \phi_{k+1}^{\mathsf{nat}}$.*

Proof

$$[\![\mathsf{S}]\!](\phi) = \chi_{F_{\mathsf{nat}}} + \underline{\mathrm{size}}_{F_{\mathsf{nat}},\mathsf{nat}}(\underline{\mathrm{Inj}}^1(\phi)) = \chi_{F_{\mathsf{nat}}} + \underline{\mathrm{size}}_{F_{\mathsf{nat}},\mathsf{nat}}(\emptyset \sqcup \downarrow \phi) =$$
$$\chi_{F_{\mathsf{nat}}} + \left(\bigvee \underline{\mathrm{size}}_{t,\mathsf{nat}}[\downarrow \phi]\right) = \chi_{F_{\mathsf{nat}}} + \left(\bigvee(\downarrow \phi)\right) = \chi_{F_{\mathsf{nat}}} + \phi.$$

■

Now set $\mathsf{n} = \mathsf{S}(\ldots(\mathsf{S}\ \mathsf{Z})\ldots) : \mathsf{nat}$ (n Ss). We will show that $[\![\mathsf{n}]\!] = \phi_{n+1}^{\mathsf{nat}}$ by induction on n. For $n = 0$,

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$[\![0]\!] = \underline{C}_{F_{\mathsf{nat}}}(\underline{\mathrm{Inj}}^0 *) = \underline{C}_{F_{\mathsf{nat}}}(\{*\} \sqcup \emptyset) = \chi_{F_{\mathsf{nat}}} + \underline{\mathrm{size}}_{F_{\mathsf{nat}},\mathsf{nat}}(\{*\} \sqcup \emptyset) = \chi_{F_{\mathsf{nat}}} + (\lambda\delta.0) = \chi_{F_{\mathsf{nat}}} = \phi_1^{\mathsf{nat}}.$$

And for $n \geq 0$, we use Lemma 9 to show that

$$[\![\mathsf{n}{+}1]\!] = ([\![\mathsf{S}]\!]\,[\![\mathsf{n}]\!]) = [\![\mathsf{S}]\!](\phi_{n+1}^{\mathsf{nat}}) = \phi_{n+2}^{\mathsf{nat}}.$$

Now let us consider closed nat tree expressions built up using only nat tree and nat constructors—i.e., nat-labeled binary trees. We show that if t is such a tree, then $[\![t]\!] = \phi_{m,k}^{\mathsf{nat\ tree}}$, where $m = \bigvee\{1 + n \mid n \text{ a label in } t\}$ and k is the number of nat tree constructors in t. In the following calculations, we will save a bit of space by writing $\underline{\mathrm{size}}_F$ for $\underline{\mathrm{size}}_{F,\mathsf{nat\ tree}}$. For emp, the argument is essentially the same as for the analysis of $[\![0]\!]$, noting that $\bigvee\{1 + n \mid n \text{ a label in emp}\} = \bigvee \emptyset = 0$. For the inductive step, assume that $[\![t_i]\!] = \phi_{n_i,k_i}^{\mathsf{nat\ tree}}$, so our goal is to show that $[\![\mathsf{node}(\mathsf{n}, t_0, t_1)]\!] = \phi_{(n+1)\vee n_0 \vee n_1, 1+k_0+k_1}^{\mathsf{nat\ tree}}$:

$$\begin{aligned}
[\![\mathsf{node}(\mathsf{n}, t_0, t_1)]\!] &= \underline{C}_{F_{\mathsf{nat\ tree}}}(\underline{\mathrm{Inj}}^1([\![\mathsf{n}]\!], [\![t_0]\!], [\![t_1]\!])) \\
&= \underline{C}_{F_{\mathsf{nat\ tree}}}(\emptyset \sqcup \downarrow(\phi_{n+1}^{\mathsf{nat}}, \phi_{n_0,k_0}^{\mathsf{nat\ tree}}, \phi_{n_1,k_1}^{\mathsf{nat\ tree}})) \\
&= \chi_{F_{\mathsf{nat\ tree}}} + \underline{\mathrm{size}}_{F_{\mathsf{nat\ tree}}}(\emptyset \sqcup \downarrow(\phi_{n+1}^{\mathsf{nat}}, \phi_{n_0,k_0}^{\mathsf{nat\ tree}}, \phi_{n_1,k_1}^{\mathsf{nat\ tree}})) \\
&= \chi_{F_{\mathsf{nat\ tree}}} + \bigvee \underline{\mathrm{size}}_{\mathsf{nat}\times t\times t}[\downarrow(\phi_{n+1}^{\mathsf{nat}}, \phi_{n_0,k_0}^{\mathsf{nat\ tree}}, \phi_{n_1,k_1}^{\mathsf{nat\ tree}})].
\end{aligned}$$

If $(\phi', \phi_0', \phi_1') \in \downarrow(\phi_{n+1}^{\mathsf{nat}}, \phi_{n_0,k_0}^{\mathsf{nat\ tree}}, \phi_{n_1,k_1}^{\mathsf{nat\ tree}})$, then

$$\begin{aligned}
\underline{\mathrm{size}}_{\mathsf{nat}\times t\times t}(\phi', \phi_0', \phi_1')(\delta) &= \begin{cases} \underline{\mathrm{size}}_{\mathsf{nat}}(\phi')(\delta) + \underline{\mathrm{size}}_t(\phi_0')(\delta) + \underline{\mathrm{size}}_t(\phi_1')(\delta), & \delta = \mathsf{nat\ tree} \\ \underline{\mathrm{size}}_{\mathsf{nat}}(\phi')(\delta) \vee \underline{\mathrm{size}}_t(\phi_0')(\delta) \vee \underline{\mathrm{size}}_t(\phi_1')(\delta), & \delta \neq \mathsf{nat\ tree} \end{cases} \\
&= \begin{cases} \phi'(\mathsf{nat\ tree}) + \phi_0'(\mathsf{nat\ tree}) + \phi_1'(\mathsf{nat\ tree}), & \delta = \mathsf{nat\ tree} \\ \phi'(\delta) \vee \phi_0'(\delta) \vee \phi_1'(\delta), & \delta \neq \mathsf{nat\ tree} \end{cases}
\end{aligned}$$

and hence the computation of $[\![\mathsf{node}(\mathsf{n}, t_0, t_1)]\!](\delta)$ proceeds as

$$\begin{aligned}
&= \begin{cases} 1 + \bigvee \left\{ \begin{array}{l} \phi'(\mathsf{nat\ tree}) + \phi_0'(\mathsf{nat\ tree}) + \phi_1'(\mathsf{nat\ tree}) \mid \\ \quad (\phi', \phi_0', \phi_1') \in \downarrow(\phi_{n+1}^{\mathsf{nat}}, \phi_{n_0,k_0}^{\mathsf{nat\ tree}}, \phi_{n_1,k_1}^{\mathsf{nat\ tree}}) \end{array} \right\}, & \delta = \mathsf{nat\ tree} \\ \bigvee \left\{ \begin{array}{l} \phi'(\delta) \vee \phi_0'(\delta) \vee \phi_1'(\delta) \mid \\ (\phi', \phi_0', \phi_1') \in \downarrow(\phi_{n+1}^{\mathsf{nat}}, \phi_{n_0,k_0}^{\mathsf{nat\ tree}}, \phi_{n_1,k_1}^{\mathsf{nat\ tree}}) \end{array} \right\}, & \delta \neq \mathsf{nat\ tree} \end{cases} \\
&= \begin{cases} 1 + k_0 + k_1, & \delta = \mathsf{nat\ tree} \\ (n+1) \vee n_0 \vee n_1, & \delta = \mathsf{nat} \\ 0, & \text{otherwise} \end{cases} \\
&= \phi_{(n+1)\vee n_0 \vee n_1, 1+k_0+k_1}^{\mathsf{nat\ tree}}(\delta).
\end{aligned}$$

We have simplified descriptions of recurrences that are analogous to those of Proposition 7:

Proposition 9.

1. *If* $f\ \phi = [\![\mathsf{fold}_{\mathsf{nat}}\ x\ \mathsf{of}\ \{\mathsf{Z} \Rightarrow e_{\mathsf{Z}} \mid \mathsf{S} \Rightarrow r.e_{\mathsf{S}}\}]\!]\eta\{x \mapsto \phi\}$, *then*

$$f\ \phi_1^{\mathsf{nat}} = [\![e_{\mathsf{Z}}]\!]\eta \qquad f\ \phi_n^{\mathsf{nat}} = [\![e_{\mathsf{Z}}]\!]\eta \vee \bigvee\{[\![e_{\mathsf{S}}]\!]\eta\{r \mapsto \phi_j^{\mathsf{nat}}\} \mid j < n\} = [\![e_{\mathsf{Z}}]\!]\eta \vee [\![e_{\mathsf{S}}]\!]\eta\{r \mapsto \phi_{n-1}^{\mathsf{nat}}\} \qquad (n > 1)$$

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

2. *If* $f\ \phi = [\![\mathsf{fold}_{\mathsf{nat\ tree}}\ x\ \mathsf{of}\ \{\mathsf{emp} \Rightarrow e_{\mathsf{emp}} \mid \mathsf{node} \Rightarrow (x, r_0, r_1).e_{\mathsf{node}}\}]\!]\eta\{x \mapsto \phi\}$, *then*

$$f\ \phi^{\mathsf{nat\ tree}}_{n,1} = [\![e_{\mathsf{emp}}]\!]\eta$$

$$f\ \phi^{\mathsf{nat\ tree}}_{n,k} = [\![e_{\mathsf{emp}}]\!]\eta \vee \bigvee \Big\{ [\![e_{\mathsf{node}}]\!]\eta\{x, r_0, r_1 \mapsto \phi^{\mathsf{nat}}_{n'}, f\ \phi^{\mathsf{nat\ tree}}_{n_0,k_0}, f\ \phi^{\mathsf{nat\ tree}}_{n_1,k_1}\} \ (n > 1) \mid n' \vee n_0 \vee n_1 \leq n, 1 + k_0 + k_1 \leq k \Big\}$$

7.3.2 *Example: summing the nodes of a* `nat tree`

Let us use this model to analyze the function `sumtree` : `nat tree` → `nat` that sums the nodes of a `nat tree`. Its definition is given in Figure 22, along the relevant extracted recurrences. An informal analysis might proceed as follows. Because the cost of `sumtree` depends on both the cost and size of the result of `plus` as well as the size of the results of the recursive calls, we must extract recurrences for all of these. If $S_{\mathtt{plus}}(m, n)$ and $T_{\mathtt{plus}}(m, n)$ are the size of the result and the cost of $\mathtt{plus}(\underline{m-1}, \underline{n-1})$, respectively (recall from Figure 4 that $\underline{n}$ is the source language numeral for n), then an informal analysis yields the recurrences

$$S_{\mathtt{plus}}(1, n) = n \qquad T_{\mathtt{plus}}(1, n) = 1$$
$$S_{\mathtt{plus}}(m, n) = 1 + S_{\mathtt{plus}}(m-1, n) \qquad T_{\mathtt{plus}}(m, n) = 1 + T_{\mathtt{plus}}(m-1, n).$$

Similarly, if $S_{\mathtt{st}}(n, k)$ and $T_{\mathtt{st}}(n, k)$ are the size of the result and the cost of $\mathtt{sumtree}(t)$ when t has maximum label size n and size k, we end up with the recurrences

$$S_{\mathtt{st}}(n, 1) = 1$$
$$S_{\mathtt{st}}(n, k) = \bigvee \{S_{\mathtt{plus}}(n, S_{\mathtt{plus}}(S_{\mathtt{st}}(n, k_0), S_{\mathtt{st}}(n, k_1))) \mid k_0 + k_1 < k\}$$

and

$$T_{\mathtt{st}}(n, 1) = 1$$
$$T_{\mathtt{st}}(n, k) = \bigvee \{T_{\mathtt{st}}(n, k_0) + T_{\mathtt{st}}(n, k_1) + T_{\mathtt{plus}}(S_{\mathtt{st}}(n, k_0), S_{\mathtt{st}}(n, k_1)) + T_{\mathtt{plus}}(n, S_{\mathtt{plus}}(S_{\mathtt{st}}(n, k_0), S_{\mathtt{st}}(n, k_1))) \mid k_0 + k_1 < k\}.$$

To solve these recurrences, one would first use any standard technique to conclude that $S_{\mathtt{plus}}(m, n) = m + n - 1$ and $T_{\mathtt{plus}}(m, n) = m$ to simplify the recurrence clauses for $S_{\mathtt{st}}$ and $T_{\mathtt{st}}$, then establish bounds on the latter by induction. However, the solution of the recurrences is not our focus here, but rather the justified extraction of them.

Now let us turn to our formal analysis. Set $\tilde{S}_{\mathsf{plus}}(\phi, \phi') = ([\![\mathsf{plus}]\!]\ \phi\ \phi')_p$. Then making use of Proposition 9, $\tilde{S}_{\mathsf{plus}}(\phi^{\mathsf{nat}}_1, \phi') = \phi'$ and for $m > 1$,

$$\begin{aligned}\tilde{S}_{\mathsf{plus}}(\phi^{\mathsf{nat}}_m, \phi') &= \phi' \vee [\![\mathsf{S}(r_p)]\!]\{r \mapsto [\![\mathsf{plus}]\!]\ \phi^{\mathsf{nat}}_{m-1}\ \phi'\} \\ &= \phi' \vee (\chi_{F_{\mathsf{nat}}} + ([\![\mathsf{plus}]\!]\ \phi^{\mathsf{nat}}_{m-1}\ \phi')_p) \qquad \text{(Proposition 9)} \\ &= \phi' \vee (\chi_{F_{\mathsf{nat}}} + \tilde{S}_{\mathsf{plus}}(\phi^{\mathsf{nat}}_{m-1}, \phi'))\end{aligned}$$

This recursive description of $\tilde{S}_{\mathsf{plus}}$ is sufficient to prove that $\tilde{S}_{\mathsf{plus}}(\phi^{\mathsf{nat}}_m, \phi') \geq \phi'$, and so we can conclude the reasoning with

$$\tilde{S}_{\mathsf{plus}}(\phi^{\mathsf{nat}}_m, \phi') = \chi_{F_{\mathsf{nat}}} + \tilde{S}_{\mathsf{plus}}(\phi^{\mathsf{nat}}_{m-1}, \phi')$$

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

and so in particular

$$\tilde{S}_{\mathsf{plus}}(\phi_1^{\mathsf{nat}}, \phi_n^{\mathsf{nat}}) = \phi_n^{\mathsf{nat}} \qquad \tilde{S}_{\mathsf{plus}}(\phi_m^{\mathsf{nat}}, \phi_n^{\mathsf{nat}}) = \chi_{F_{\mathsf{nat}}} + \tilde{S}_{\mathsf{plus}}(\phi_{m-1}^{\mathsf{nat}}, \phi_n^{\mathsf{nat}}),$$

recurrences that are equivalent to those derived informally. The analysis of $\tilde{T}_{\mathsf{plus}}(\phi, \phi') = (\llbracket \mathsf{plus} \rrbracket\ \phi\ \phi')_c$ is similar and results in the recurrence

$$\tilde{T}_{\mathsf{plus}}(\phi_1^{\mathsf{nat}}, \phi_n^{\mathsf{nat}}) = 1 \qquad \tilde{T}_{\mathsf{plus}}(\phi_m^{\mathsf{nat}}, \phi_n^{\mathsf{nat}}) = 1 + \tilde{T}_{\mathsf{plus}}(\phi_{m-1}^{\mathsf{nat}}, \phi_n^{\mathsf{nat}}).$$

Now set $\tilde{S}_{\mathsf{st}}(\phi) = (\llbracket \mathsf{sumtree} \rrbracket\ \phi)_p$. Making use of Proposition 9, $\tilde{S}_{\mathsf{st}}(\phi_{1,k}^{\mathsf{nat\,tree}}) = \phi_1^{\mathsf{nat}}$ and for $k > 1$,

$$\begin{aligned}
\tilde{S}_{\mathsf{st}}(\phi_{n,k}^{\mathsf{nat\,tree}}) &= \phi_1^{\mathsf{nat}} \vee \bigvee \big\{(\llbracket \mathsf{plus}\, x\, (\mathsf{plus}\, r_{0p}\, r_{1p})_p \rrbracket\{x, r_i \mapsto \phi_{n'}^{\mathsf{nat}}, \llbracket \mathsf{sumtree} \rrbracket\ \phi_{n_i,k_i}^{\mathsf{nat\,tree}}\})_p \\
&\qquad\qquad \mid n' \vee n_0 \vee n_1 \leq n, k_0 + k_1 < k\big\} \\
&= \phi_1^{\mathsf{nat}} \vee \bigvee \big\{(\llbracket \mathsf{plus} \rrbracket\ \phi_{n'}^{\mathsf{nat}}(\llbracket \mathsf{plus} \rrbracket\ (\tilde{S}_{\mathsf{st}}(\phi_{n_0,k_0}^{\mathsf{nat\,tree}}))\ (\tilde{S}_{\mathsf{st}}(\phi_{n_1,k_1}^{\mathsf{nat\,tree}})))_p)_p \\
&\qquad\qquad \mid n' \vee n_0 \vee n_1 \leq n, k_0 + k_1 < k\big\} \\
&= \phi_1^{\mathsf{nat}} \vee \bigvee \big\{\tilde{S}_{\mathsf{plus}}(\phi_{n'}^{\mathsf{nat}}, \tilde{S}_{\mathsf{plus}}(\tilde{S}_{\mathsf{st}}(\phi_{n_0,k_0}^{\mathsf{nat\,tree}}), \tilde{S}_{\mathsf{st}}(\phi_{n_1,k_1}^{\mathsf{nat\,tree}}))) \\
&\qquad\qquad \mid n' \vee n_0 \vee n_1 \leq n, k_0 + k_1 < k\big\}
\end{aligned}$$

Since ϕ_1^{nat} is the bottom element of W^{nat} and we can prove from this recurrence that $\tilde{S}_{\mathsf{st}}(\phi_{n,k}^{\mathsf{nat\,tree}})$ is monotone with respect to n, we can conclude this reasoning with

$$\tilde{S}_{\mathsf{st}}(\phi_{n,k}^{\mathsf{nat\,tree}}) = \bigvee \{\tilde{S}_{\mathsf{plus}}(\phi_n^{\mathsf{nat}}, \tilde{S}_{\mathsf{plus}}(\tilde{S}_{\mathsf{st}}(\phi_{n,k_0}^{\mathsf{nat\,tree}}), \tilde{S}_{\mathsf{st}}(\phi_{n,k_1}^{\mathsf{nat\,tree}}))) \mid k_0 + k_1 < k\},$$

which is analogous to the recurrence we derived informally. The analysis of $\tilde{T}_{\mathsf{st}}(\phi_{n,k}^{\mathsf{nat\,tree}}) = (\llbracket \mathsf{sumtree} \rrbracket\ \phi_{n,k}^{\mathsf{nat\,tree}})_c$ is similar and leads to

$$\begin{aligned}
\tilde{T}_{\mathsf{st}}(\phi_{1,k}^{\mathsf{nat\,tree}}) = 1 \qquad \tilde{T}_{\mathsf{st}}(\phi_{n,k}^{\mathsf{nat\,tree}}) = \bigvee \Big\{ & 1 + \tilde{T}_{\mathsf{st}}(\phi_{n,k_0}^{\mathsf{nat\,tree}}) + \tilde{T}_{\mathsf{st}}(\phi_{n,k_1}^{\mathsf{nat\,tree}}) + \\
& \tilde{T}_{\mathsf{plus}}(\tilde{S}_{\mathsf{st}}(\phi_{n,k_0}^{\mathsf{nat\,tree}}), \tilde{S}_{\mathsf{st}}(\phi_{n,k_1}^{\mathsf{nat\,tree}})) + \\
& \tilde{T}_{\mathsf{plus}}(\phi_n^{\mathsf{nat}}, \tilde{S}_{\mathsf{plus}}(\tilde{S}_{\mathsf{st}}(\phi_{n,k_0}^{\mathsf{nat\,tree}}), \tilde{S}_{\mathsf{st}}(\phi_{n,k_1}^{\mathsf{nat\,tree}}))) \\
& \mid k_0 + k_1 < k \Big\}
\end{aligned}$$

As a final note, in order to obtain the desired final form, we sometimes had to do some reasoning about the function on the basis of its recurrence, such as proving that the function is monotone. In fact, such reasoning is almost always required in the informal analysis as well, even though we typically gloss over such points when analyzing algorithms.

In may be helpful to contrast this analysis with the interpretation of plus and sumtree in the model of Section 7.2. Since nat values involve no other datatype constructors, the interpretation of plus is essentially just the same, only requiring less notation to write down. However, the cost component of $\llbracket \mathsf{sumtree} \rrbracket\{n/t\}$ is less helpful. Because the model of Section 7.2 only accounts for the tree constructors, it does not account for the sizes of the node labels, and so this computation includes the cost component of $\llbracket \mathsf{plus}\, x\, (\mathsf{plus}\, r_{0p}\, r_{1p})_p \rrbracket\{\infty, \ldots/x, r_0, r_1\}$ and this will result in a bound of ∞ (cf. to the occurrence of ∞ in the analysis of `mem` in the previous section, which did no harm there). This is correct as a bound. It reflects a cost analysis in which we have decided that we are counting each recursive call as a computation step, but then analyze a program in which data values whose size we ignore is the source of some recursive calls. However, this rather

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

poor choice of size for this particular context yields a very weak bound, and so shows more generally that the choice of model does really matter.

7.4 Size abstraction and polymorphism: merging the constructor-counting models

Let us make a couple of observations about the previous two sections. It seems at least intuitive that counting only the main constructors is a more abstract notion of size than counting all constructors. And it also seems that even if we are working in the model of Section 7.3, if we have a polymorphic function in hand, it ought to be analyzable by just counting main constructors. This leads to the idea that if we have a model in hand (such as counting all constructors), then at least in some cases, it ought to be possible to interpret polymorphic recurrences so that the potentials arise from a more abstract notion of size than that given by the model. We give an example of how that might be done now.

Definition. *Suppose* $\mathbf{U} = (\mathbf{U}_{sm}, \mathbf{U}_{lg}, \{D^\sigma\})$ *and* $\mathbf{U}' = (\mathbf{U}_{sm}, \mathbf{U}_{lg}, \{D'\}^\sigma)$ *are two models of the recurrence language, both based on the (same extension of the) standard type frame. We say that* $\mathbf{U}'$ is an abstraction of $\mathbf{U}$*, or* $\mathbf{U}$ is a concretization of $\mathbf{U}'$*, if for every* $\sigma \in \mathbf{U}_{sm}$ *there are functions*

$$D^\sigma \underset{\text{conc}_\sigma}{\overset{\text{abs}_\sigma}{\rightleftarrows}} D'^\sigma$$

such that for all σ*,* conc_σ *is monotone,* $\text{conc}_\sigma \circ \text{abs}_\sigma \geq \text{id}_{D^\sigma}$ *and* $\text{abs}_\sigma \circ \text{conc}_\sigma = \text{id}_{D'^\sigma}$.

Definition. *Suppose* $\mathbf{U}' = (\mathbf{U}_{sm}, \mathbf{U}_{lg}, \{D'\}^\sigma)$ *is an abstraction of* $\mathbf{U} = (\mathbf{U}_{sm}, \mathbf{U}_{lg}, \{D^\sigma\})$*. The* polymorphic abstraction of $\mathbf{U}$ relative to $\mathbf{U}'$ *is the model* $\mathbf{U} \to \mathbf{U}' = (\mathbf{U}_{sm}, \mathbf{U}_{lg}, \{B^\sigma\})$ *that is defined as follows:*

- *For* $\sigma \in \mathbf{U}_{sm}$*,* $B^\sigma = D^\sigma$*, with the semantic functions for small types taken from* $\mathbf{U}$*.*
- *For* $\tau \in \mathbf{U}_{lg} \setminus \mathbf{U}_{sm}$*,* $B^\tau = D'^\tau$*, where:*
  - *If* ρ *is quantifier-free and* $\text{fv}(\rho) \subseteq \{\alpha\}$*, then*

$$\text{dom}(\underline{\text{TyAbs}}^{\mathbf{U}\to\mathbf{U}'}_{\boldsymbol{\lambda}\sigma.\rho\{\sigma/\alpha\}}) = \{f \in \prod_{\sigma\in\mathbf{U}_{sm}} (D')^{\rho\{\sigma/\alpha\}} \mid \lambda\!\!\lambda\sigma.\text{abs}_{\rho\{\sigma/\alpha\}}(f\,\sigma) \in \text{dom}(\underline{\text{TyAbs}}^{\mathbf{U}'}_{\boldsymbol{\lambda}\sigma.\rho\{\sigma/\alpha\}})\}$$

$$\underline{\text{TyAbs}}_{\boldsymbol{\lambda}\sigma.\rho\{\sigma/\alpha\}}(f) = \underline{\text{TyAbs}}^{\mathbf{U}'}_{\boldsymbol{\lambda}\sigma.\rho\{\sigma/\alpha\}}(\lambda\!\!\lambda\sigma.\text{abs}_{\rho\{\sigma/\alpha\}}(f\,\sigma))$$

$$\underline{\text{TyApp}}_{\boldsymbol{\lambda}\sigma.\rho\{\sigma/\alpha\}}(f) = \lambda\!\!\lambda\sigma.\text{conc}_{\rho\{\sigma/\alpha\}}(\underline{\text{TyApp}}^{\mathbf{U}'}_{\boldsymbol{\lambda}\sigma.\rho\{\sigma/\alpha\}} f\,\sigma)$$

  - *If* τ *is not quantifier-free and* $\text{fv}(\tau) \subseteq \{\alpha\}$*, then we take* $\underline{\text{TyAbs}}_{\boldsymbol{\lambda}\sigma.\tau\{\sigma/\alpha\}} = \underline{\text{TyAbs}}^{\mathbf{U}'}_{\boldsymbol{\lambda}\sigma.\tau\{\sigma/\alpha\}}$ *and* $\underline{\text{TyApp}}_{\boldsymbol{\lambda}\sigma.\tau\{\sigma/\alpha\}} = \underline{\text{TyApp}}^{\mathbf{U}'}_{\boldsymbol{\lambda}\sigma.\tau\{\sigma/\alpha\}}$.

Proposition 10.

1. *If* $\mathbf{U}$ *and* $\mathbf{U}'$ *are applicative structures, then* $\mathbf{U} \to \mathbf{U}'$ *is an applicative structure.*
2. *If* $\mathbf{U}$ *and* $\mathbf{U}'$ *are premodels such that whenever* $\Gamma \vdash e : \rho$ *and* η *is a* Γ*-environment,* $\lambda\!\!\lambda\sigma.[\![\text{abs}_{\rho\{\sigma/\alpha\}}(e)]\!]\eta\{\alpha \mapsto \sigma\} \in \text{dom}\,\underline{\text{TyAbs}}^{\mathbf{U}'}$*, then* $\mathbf{U} \to \mathbf{U}'$ *is a premodel.*

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\begin{aligned}
\mathrm{abs}_\sigma &: W^\sigma \to V^\sigma & \mathrm{conc}_\sigma &: V^\sigma \to W^\sigma \\
\mathrm{abs}_{\mathsf{unit}}(*) &= * & \mathrm{conc}_{\mathsf{unit}}(*) &= * \\
\mathrm{abs}_{\mathsf{C}}(n) &= n & \mathrm{conc}_{\mathsf{C}}(n) &= n \\
\mathrm{abs}_{\mu t.F}(\phi) &= \phi(\mu t.F) & \mathrm{conc}_{\mu t.F}(n) &= \lambda\!\!\lambda\, \delta.\, \begin{cases} n, & \delta = \mu t.F \\ \infty, & \delta \neq \mu t.F \end{cases} \\
\mathrm{abs}_{\sigma_0+\sigma_1}(X_0 \sqcup X_1) &= \downarrow \mathrm{abs}_{\sigma_0}[X_0] \sqcup \downarrow \mathrm{abs}_{\sigma_1}[X_1] & \mathrm{conc}_{\sigma_0+\sigma_1}(Y_0 \sqcup Y_1) &= \downarrow \mathrm{conc}_{\sigma_0}[Y_0] \sqcup \downarrow \mathrm{conc}_{\sigma_1}[Y_1] \\
\mathrm{abs}_{\sigma_0\times\sigma_1}(x_0, x_1) &= (\mathrm{abs}_{\sigma_0}(x_0), \mathrm{abs}_{\sigma_1}(x_1)) & \mathrm{conc}_{\sigma_0\times\sigma_1}(y_0, y_1) &= (\mathrm{conc}_{\sigma_0}(y_0), \mathrm{conc}_{\sigma_1}(y_1)) \\
\mathrm{abs}_{\rho\to\sigma}(f) &= \mathrm{abs}_\sigma \circ f \circ \mathrm{conc}_\rho & \mathrm{conc}_{\rho\to\sigma}(f) &= \mathrm{conc}_\sigma \circ f \circ \mathrm{abs}_\rho
\end{aligned}$$

Fig. 23: Abstraction and concretization functions that relate the all-constructor (concrete) and main-constructor (abstract) models.

Proof The only nontrivial verification is that when ρ is quantifier-free and $\mathrm{fv}(\rho) \subseteq \{\alpha\}$, $\underline{\mathrm{TyApp}}_{\lambda\!\!\lambda\sigma.\rho\{\sigma/\alpha\}}(\underline{\mathrm{TyAbs}}_{\lambda\!\!\lambda\sigma.\rho\{\sigma/\alpha\}} f) \geq f$:

$$\begin{aligned}
&\underline{\mathrm{TyApp}}_{\lambda\!\!\lambda\sigma.\rho\{\sigma/\alpha\}}(\underline{\mathrm{TyAbs}}_{\lambda\!\!\lambda\sigma.\rho\{\sigma/\alpha\}} f)\,\sigma \\
&\quad = \underline{\mathrm{TyApp}}_{\lambda\!\!\lambda\sigma.\rho\{\sigma/\alpha\}}(\underline{\mathrm{TyAbs}}^{\mathbf{U}'}_{\lambda\!\!\lambda\sigma.\rho\{\sigma/\alpha\}}(\lambda\!\!\lambda\sigma.\mathrm{abs}_{\rho\{\sigma/\alpha\}}(f\,\sigma)))\,\sigma \\
&\quad = (\lambda\!\!\lambda\sigma.\mathrm{conc}_{\rho\{\sigma/\alpha\}}(\underline{\mathrm{TyApp}}^{\mathbf{U}'}_{\lambda\!\!\lambda\sigma.\rho\{\sigma/\alpha\}}(\underline{\mathrm{TyAbs}}^{\mathbf{U}'}_{\lambda\!\!\lambda\sigma.\rho\{\sigma/\alpha\}}(\lambda\!\!\lambda\sigma.\mathrm{abs}_{\rho\{\sigma/\alpha\}}(f\,\sigma)))\,\sigma))\,\sigma \\
&\quad = \mathrm{conc}_{\rho\{\sigma/\alpha\}}(\underline{\mathrm{TyApp}}^{\mathbf{U}'}_{\lambda\!\!\lambda\sigma.\rho\{\sigma/\alpha\}}(\underline{\mathrm{TyAbs}}^{\mathbf{U}'}_{\lambda\!\!\lambda\sigma.\rho\{\sigma/\alpha\}}(\lambda\!\!\lambda\sigma.\mathrm{abs}_{\rho\{\sigma/\alpha\}}(f\,\sigma)))\,\sigma) \\
&\quad \geq \mathrm{conc}_{\rho\{\sigma/\alpha\}}((\lambda\!\!\lambda\sigma.\mathrm{abs}_{\rho\{\sigma/\alpha\}}(f\,\sigma))\,\sigma) \\
&\quad \geq \mathrm{conc}_{\rho\{\sigma/\alpha\}}(\mathrm{abs}_{\rho\{\sigma/\alpha\}}(f\,\sigma)) \\
&\quad \geq f\,\sigma.
\end{aligned}$$

■

As an example, we define abstraction and concretization functions in Figure 23 that show that the main constructor counting model **V** from Section 7.2 is an abstraction of the all-constructor counting model **W** from Section 7.3.

Proposition 11.

1. abs_σ *and* conc_σ *are monotone for all* σ.
2. $\mathrm{abs}_\sigma \circ \mathrm{conc}_\sigma = \mathrm{id}$ *and* $\mathrm{conc}_\sigma \circ \mathrm{abs}_\sigma \geq \mathrm{id}$.

Proof

1. By induction on σ.
2. By induction on σ; we just do $\sigma = \sigma_0 + \sigma_1$. Let us write abs for $\mathrm{abs}_{\sigma_0+\sigma_1}$, abs_i for abs_{σ_i}, and similarly for conc. To see that $\mathrm{abs} \circ \mathrm{conc} = \mathrm{id}$, notice that $(\mathrm{abs} \circ \mathrm{conc})(Y_0 \sqcup Y_1) = \downarrow \mathrm{abs}_0[\downarrow \mathrm{conc}_0[Y_0]] \sqcup \downarrow \mathrm{abs}_1[\downarrow \mathrm{conc}_1[Y_1]]$, so if $a' \in (\mathrm{abs} \circ \mathrm{conc})(Y_0 \sqcup Y_1)$, then there are i, b, and $a \in Y_i$ such that $a' \leq \mathrm{abs}_i(b)$ and $b \leq \mathrm{conc}_i(a)$, and hence $a' \leq \mathrm{abs}_i(\mathrm{conc}_i(a)) = a$ (by monotonicity and the induction hypothesis). But since Y_i is downward closed, $a' \in Y_i$, so $(\mathrm{abs} \circ \mathrm{conc})(Y_0 \sqcup Y_1) \subseteq Y_0 \sqcup Y_1$. To see that $\mathrm{abs} \circ \mathrm{conc} \geq \mathrm{id}$, notice that if $a \in Y_0 \sqcup Y_1$, then $a \in Y_i$ for some i, and hence $a = \mathrm{abs}_i(\mathrm{conc}_i(a)) \in \downarrow \mathrm{abs}_0[\downarrow \mathrm{conc}_0[Y_0]] \sqcup \downarrow \mathrm{abs}_1[\downarrow \mathrm{conc}_1[Y_1]] = (\mathrm{abs} \circ \mathrm{conc})(Y_0 \sqcup Y_1)$, so $Y_0 \sqcup Y_1 \subseteq (\mathrm{abs} \circ \mathrm{conc})(Y_0 \sqcup Y_1)$.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

To see that $\mathrm{conc} \circ \mathrm{abs} \geq \mathrm{id}$, suppose $b \in X_i$. Then by the induction hypothesis $b \leq (\mathrm{conc}_i \circ \mathrm{abs}_i)(b)$, and by unraveling the definition, $(\mathrm{conc}_i \circ \mathrm{abs}_i)(b) \in \downarrow \mathrm{conc}_i[\downarrow \mathrm{abs}_i[X_i]]$. Since $\downarrow \mathrm{conc}_i[\downarrow \mathrm{abs}_i[X_i]]$ is downward closed, $b \in \downarrow \mathrm{conc}_i[\downarrow \mathrm{abs}_i[X_i]] \subseteq (\mathrm{conc} \circ \mathrm{abs})(X_0 \sqcup X_1)$.

■

Proposition 12. $\mathbf{W} \to \mathbf{V}$ *is a model.*

Proof From Propositions 10 and 11 and the fact that $\underline{\mathrm{TyAbs}}^{\mathbf{V}}$ is total. ■

The definition of the abstraction and concretization functions in Figure 23 looks fairly canonical, so a natural question is whether for any two models of the recurrence language one can extend given functions on the interpretations of base types to all small types. In fact these definitions are an instance of a general pattern, but to state the pattern we will need a few definitions. A 2-category is a generalization of a category with a notion of morphism-between-morphism: if X and Y are objects, and $f, g : X \longrightarrow Y$ are morphisms, then we will write $f \leq_{\mathscr{C}} g : X \longrightarrow Y$ for a 2-cell from f to g. We will mainly consider the 2-category **Preorder**, whose objects X, Y are preordered sets, whose morphisms $f : X \longrightarrow Y$ are monotone functions, and whose 2-cells $f \leq g : X \longrightarrow Y$ are bounds $\forall x : X. f(x) \leq_Y g(x)$. We will also need $\mathbf{Preorder}^{op}$ (the 1-cell dual of **Preorder**): the objects are again preorders, a 1-cell $X \longrightarrow_{\mathbf{Preorder}^{op}} Y$ in $\mathbf{Preorder}^{op}$ is a 1-cell in $Y \longrightarrow_{\mathbf{Preorder}} X$, i.e. a monotone function $Y \to X$, but the 2-cells $f \leq_{Preorder^{op}} g : X \longrightarrow_{Preorder^{op}} Y$ are still the 2-cells $f \leq_{\mathbf{Preorder}} g : Y \longrightarrow_{\mathbf{Preorder}} X$, i.e. $\forall y : Y. f(y) \leq_X g(y)$. A standard construction is to take the cartesian product of two 2-categories, where the objects, 1-cells, and 2-cells are given pointwise; in particular we will consider $\mathbf{Preorder} \times \mathbf{Preorder}$ and $\mathbf{Preorder}^{op} \times \mathbf{Preorder}$. A 2-functor $F : \mathscr{C} \to \mathscr{D}$ between 2-categories acts on objects, 1-cells (preserving identity and composition either strictly or up to 2-cell isomorphism), and 2-cells. For example, a (strict) 2-functor $F : \mathbf{Preorder} \to \mathbf{Preorder}$ consists of (0) for each preorder X, a preorder $F(X)$; (1) for each monotone function $f : X \to Y$, a monotone function $F(f) : F(X) \to F(Y)$ such that $F(id) = id$ and $F(g \circ f) = F(g) \circ F(f)$; (2) if $\forall x : X. f(x) \leq_Y g(x)$ then $\forall w : F(X). F(f)w \leq_{F(Y)} F(g)w$. I.e. F sends preorders to preorders and monotone functions to monotone functions, in such a way that if g bounds f then $F(g)$ bounds $F(f)$.

An abstract interpretation in the sense above is often called a *Galois insertion*, which is a *reflection in* **Preorder**: a (strict) reflection of A into C consists of a pair of 1-cells $\mathrm{abs} \dashv \mathrm{conc}$ where $\mathrm{abs} : C \to A$ and $\mathrm{conc} : A \to C$, with an equality $\mathrm{abs} \circ \mathrm{conc} = id_A$ and a 2-cell $id_C \leq_C \mathrm{conc} \circ \mathrm{abs}$. A standard observation is that *any* 2-functor $F : \mathscr{C} \to \mathscr{D}$ preserves reflections (this is used, for example, in domain theory Smyth & Plotkin, 1982): if $\mathrm{abs} \dashv \mathrm{conc}$ is a reflection then $F(\mathrm{abs}) \dashv F(\mathrm{conc})$ is a reflection between $F(C)$ and $F(A)$. Applying F to the equality $\mathrm{abs} \circ \mathrm{conc} = id_A$ and using strict preservation of identity and composition gives $F(\mathrm{abs}) \circ F(\mathrm{conc}) = id_{F(A)}$, and using the action on 2-cells of F on $id_C \leq_C \mathrm{conc} \circ \mathrm{abs}$ (and again preservation of identity and composition) gives $id_{F(C)} \leq_{F(C)} F(\mathrm{conc}) \circ F(\mathrm{abs})$.

This all means that we can lift the abstraction and concretization from base types to any type constructor that extends to a 2-functor. The product of preorders $X \times Y$ is the action on objects of a functor $\mathbf{Preorder} \times \mathbf{Preorder} \to \mathbf{Preorder}$, where the action on

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\begin{aligned}\texttt{rev} = \lambda xs.\texttt{let}\ \texttt{rev}' = \lambda xs.\texttt{fold}_{\alpha\ \texttt{list}}\ xs\ \texttt{of}\quad & \texttt{nil} \Rightarrow \lambda zs.zs \\ & \mid \texttt{cons} \Rightarrow (x,r).\lambda zs.(\texttt{force}\ r)(\texttt{cons}(x,zs)) \\ \texttt{in}\ \texttt{rev}'\ xs\ \texttt{nil}\end{aligned}$$

$$\mathsf{rev}' = \Lambda\alpha.\lambda xs.\mathsf{fold}_{\alpha\ \mathsf{list}}\ xs\ \mathsf{of}\ \left\{\begin{array}{l}\mathsf{nil} \Rightarrow (1, \lambda zs.(0, zs)) \\ \mid \mathsf{cons} \Rightarrow (x,r).(1, \lambda zs.r_c +_c r_p\ (\mathsf{cons}(x,zs)))\end{array}\right\}$$

Fig. 24: Linear-time list reversal and its extracted recurrences.

maps $f_0 : X_0 \to X_0'$ and $g : X_1 \to X_1'$ is given by

$$f_0 \times f_1 : X_0 \times X_1 \to X_0' \times X_1' := z \mapsto \langle f_0(\pi_0 z), f_1(\pi_1 z)\rangle$$

This acts on 2-cells (preserves bounds) because pairing and application are monotone operations. To show that it preserves composition, we need a full β-reduction equation and to show that it preserves identity, we also need the corresponding η/surjective pairing equation. However, these are true for the standard cartesian product of preorders. A reflection in **Preorder** $\times$ **Preorder** is a pair of reflections for each component. Unwinding these definitions gives the definitions of $\mathrm{abs}_{\sigma_0\times\sigma_1}$ and $\mathrm{conc}_{\sigma_0\times\sigma_1}$ in Figure 23.

The case of sums is more interesting. The standard coproduct of preorders $X + Y$ is the disjoint union $X \sqcup Y$ ordered as defined above. This extends to a 2-functor **Preorder** $\times$ **Preorder** $\to$ **Preorder** with $f_0 + f_1$ defined via case-analysis. This is bound-preserving because the branches of a case-analysis (on the standard coproduct in preorders) are a monotone position and preserves identity/composition if we have $\beta\eta$ equations for case-analysis, which $X + Y$ does.

In the models under consideration, we do not define $D^{\sigma_0+\sigma_1}$ to be $D^{\sigma_0} + D^{\sigma_1}$, but $\mathscr{O}(D^{\sigma_0} + D^{\sigma_1})$. However, it is also the case that $\mathscr{O}$ is a 2-functor **Preorder** $\to$ **Preorder**: $\mathscr{O}f : \mathscr{O}X \to \mathscr{O}Y$ is $\downarrow\{f(x) : x \in X\}$, which preserves bounds and identities and compositions. The composition of 2-functors is again a 2-functor, so $\mathscr{O}(- + -) :$ **Preorder** $\times$ **Preorder** $\to$ **Preorder** is as well, and unwinding definitions gives $\mathrm{abs}_{\sigma_0+\sigma_1}$ and $\mathrm{conc}_{\sigma_0+\sigma_1}$ from Figure 23.

For functions, the preorder of pointwise-ordered monotone maps $X \to Y$ extends to a mixed-variance 2-functor **Preorder**^{op} $\times$ **Preorder** $\to$ **Preorder**, with functorial action given by pre- and post-composition. Moreover, a reflection abs $\dashv$ conc in **Preorder** is a reflection conc $\dashv$ abs in **Preorder**^{op} with the roles of concretization and abstraction exchanged. This unpacks to the definitions of $\mathrm{abs}_{\rho\to\sigma}$ and $\mathrm{conc}_{\rho\to\sigma}$ in Figure 23, where abstraction precomposes with concretization, and vice versa.

Thus, while our general definition of model does not require types to be interpreted as 2-functors—for example, being a model does not require the η law for pairs that ensures preservation of identities—a number of more specific models will have this form, and thus admit the same definition of relativized model, given abstraction and concretization for base/inductive types. For example, we may freely apply in the interpretation of any type constructor, e.g., defining $D^{\sigma_0\times\sigma_1}$ to be $\mathscr{O}(D^{\sigma_0} \times D^{\sigma_1})$ for more precision.

7.4.1 Example: list reverse

To get a sense of how polymorphic abstraction behaves, let us analyze the polymorphic linear-time list reverse function given in Figure 24 in the model $\mathbf{W} \to \mathbf{V}$. We choose

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

this model because on the one hand $\mathbf{W}$ provides enough information for analyzing monomorphic functions like `sumtree` that depend on more than just the usual notion of size, yet we still want to analyze a polymorphic function like list reversal in terms of list length, ignoring any information about the elements of the argument list. Since polymorphism in the source language arises only via let-bindings, the recurrence for rev' that is given is the recurrence that is substituted for for rev' according to the definition of extraction for `let`-expressions. A typical informal analysis of `rev` would really analyze $\mathtt{rev}'$ and might define $S(n,m)$ and $T(n,m)$ to be the size and cost of $\mathtt{rev}'\ xs\ ys$ when xs and ys have length n and m, respectively. One would then observe that S and T satisfy the recurrences

$$\begin{aligned} S(1,m) &= m & T(1,m) &= 1 \\ S(n,m) &= S(n-1,m+1) & T(n,m) &= 1 + T(n-1,m) \end{aligned}$$

from which one establishes the $O(n)$ bound on cost.

Just as with our other models, to analyze rev, we must consider its instantiation at some arbitrary small type σ. In the model $\mathbf{W}$, this would entail understanding how to compute $\underline{\mathrm{Fold}}^{\mathbf{W}}\ s\ \phi$ for arbitrary ϕ, which would be defined in terms of all $\phi' \leq \phi$. The key point of $\mathbf{W} \to \mathbf{V}$ is that while we cannot avoid considering the instantiation of rev at arbitrary σ, we only need to know how to compute $\underline{\mathrm{Fold}}^{\mathbf{W}}\ s\ \phi$ for those ϕ that are the concretizations of values in $V^{\sigma\ \mathsf{list}}$. To see this, let us define $\phi_n^{\sigma\ \mathsf{list}} = \mathrm{conc}_{\sigma\ \mathsf{list}}(n)$—observe that $\phi_n^{\sigma\ \mathsf{list}}$ maps σ list to n and all other datatypes to ∞—and then compute rev', where we write $f_\sigma\ \phi$ for $[\![\mathsf{fold}_{\sigma\ \mathsf{list}}\ xs\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \cdots \mid \mathsf{cons} \Rightarrow \cdots\}]\!]\eta\{xs \mapsto \phi\}$:

$$\begin{aligned} [\![\mathsf{rev}']\!] &= \underline{\mathrm{TyAbs}}(\lambda\!\!\lambda\sigma.\lambda\!\!\lambda\phi^{W^{\sigma\ \mathsf{list}}}.[\![\mathsf{fold}_{\alpha\ \mathsf{list}}\ xs\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \cdots \mid \mathsf{cons} \Rightarrow \cdots\}]\!]\{\alpha \mapsto \sigma, xs \mapsto \phi\}) \\ &= \underline{\mathrm{TyAbs}}(\lambda\!\!\lambda\sigma.\lambda\!\!\lambda\phi^{W^{\sigma\ \mathsf{list}}}.[\![\mathsf{fold}_{\sigma\ \mathsf{list}}\ xs\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \cdots \mid \mathsf{cons} \Rightarrow \cdots\}]\!]\{xs \mapsto \phi\}) \\ &= \underline{\mathrm{TyAbs}}(\lambda\!\!\lambda\sigma.\lambda\!\!\lambda\phi^{W^{\sigma\ \mathsf{list}}}.f_\sigma\ \phi) \\ &= \lambda\!\!\lambda\sigma.\mathrm{abs}(\lambda\!\!\lambda\phi^{W^{\sigma\ \mathsf{list}}}.f_\sigma\ \phi) \\ &= \lambda\!\!\lambda\sigma.\lambda\!\!\lambda n^{V^{\sigma\ \mathsf{list}}}.\mathrm{abs}(f_\sigma\ \phi_n^{\sigma\ \mathsf{list}}) \\ &= \lambda\!\!\lambda\sigma.\lambda\!\!\lambda n^{V^{\sigma\ \mathsf{list}}}.(\_, \mathrm{abs}(f_\sigma\ \phi_n^{\sigma\ \mathsf{list}})_p) \\ &= \lambda\!\!\lambda\sigma.\lambda\!\!\lambda n^{V^{\sigma\ \mathsf{list}}}.(\_, \lambda\!\!\lambda m^{V^{\sigma\ \mathsf{list}}}.\mathrm{abs}((f_\sigma\ \phi_n^{\sigma\ \mathsf{list}})_p\ \phi_m^{\sigma\ \mathsf{list}})) \\ &= \lambda\!\!\lambda\sigma.\lambda\!\!\lambda n^{V^{\sigma\ \mathsf{list}}}.(\_, \lambda\!\!\lambda m^{V^{\sigma\ \mathsf{list}}}.(\_, \mathrm{abs}((f_\sigma\ \phi_n^{\sigma\ \mathsf{list}})_p\ \phi_m^{\sigma\ \mathsf{list}})_p)) \end{aligned}$$

When restricted to concretizations of abstract values, $\underline{\mathrm{Fold}}^{\mathbf{W}}$ is straightforward to compute.

Proposition 13. *If* $f\ n = [\![\mathsf{fold}_{\sigma\ \mathsf{list}}\ y\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons} \Rightarrow (x,r).e_{\mathsf{cons}}\}]\!]\eta\{y \mapsto \phi_n^{\sigma\ \mathsf{list}}\}^{\mathbf{W}}$, *then*

$$\begin{aligned} f\ 1 &= [\![e_{\mathsf{nil}}]\!]\eta \\ f\ n &= [\![e_{\mathsf{nil}}]\!]\eta \vee [\![e_{\mathsf{cons}}]\!]\eta\{x, r \mapsto \infty^\sigma, f(n-1)\} \qquad (n > 1). \end{aligned}$$

With this in mind, set $\tilde{S}(n,m) = \mathrm{abs}((f_\sigma\ \phi_n^{\sigma\ \mathsf{list}})_p\ \phi_m^{\sigma\ \mathsf{list}})_p$. Our goal is to write a recurrence for $\tilde{S}(n,m)$. We start with

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\begin{aligned}\tilde{S}(1,m) &= \mathrm{abs}((f_\sigma\ \phi_1^{\sigma\ \mathsf{list}})_p\ \phi_m^{\sigma\ \mathsf{list}})_p \\ &= \mathrm{abs}((\llbracket(1,\lambda zs.(0,zs))\rrbracket)_p\ \phi_m^{\sigma\ \mathsf{list}})_p \\ &= \mathrm{abs}(\lambda\!\!\lambda\phi.(0,\phi)\ \phi_m^{\sigma\ \mathsf{list}})_p \\ &= \mathrm{abs}(0,\phi_m^{\sigma\ \mathsf{list}})_p \\ &= \mathrm{abs}(\phi_m^{\sigma\ \mathsf{list}}) \\ &= m.\end{aligned}$$

To compute $\tilde{S}(n,m)$ for $n > 1$, we first compute

$$\begin{aligned}f\ \phi_n^{\sigma\ \mathsf{list}} &= \llbracket(1,\lambda zs.(0,zs))\rrbracket \vee \llbracket(1,\lambda zs.r_c +_c r_p\ (\mathsf{cons}((x,zs))))\rrbracket\{x,r\mapsto\infty, f\ \phi_{n-1}^{\sigma\ \mathsf{list}}\} \\ &= (1,\lambda\!\!\lambda\phi.(0,\phi) \vee \lambda\!\!\lambda\phi.(f\ \phi_{n-1}^{\sigma\ \mathsf{list}})_c +_c (f\ \phi_{n-1}^{\sigma\ \mathsf{list}})_p(\chi_{F_{\sigma\ \mathsf{list}}} + \phi)) \\ &= (1,\lambda\!\!\lambda\phi.(0,\phi) \vee (f\ \phi_{n-1}^{\sigma\ \mathsf{list}})_c +_c (f\ \phi_{n-1}^{\sigma\ \mathsf{list}})_p(\chi_{F_{\sigma\ \mathsf{list}}} + \phi))\end{aligned}$$

and so

$$\begin{aligned}\left((f\ \phi_n^{\sigma\ \mathsf{list}})_p\ \phi_m^{\sigma\ \mathsf{list}}\right)_p &= \left((0,\phi_m^{\sigma\ \mathsf{list}}) \vee (f\ \phi_{n-1}^{\sigma\ \mathsf{list}})_c +_c (f\ \phi_{n-1}^{\sigma\ \mathsf{list}})_p(\chi_{F_{\sigma\ \mathsf{list}}} + \phi_m^{\sigma\ \mathsf{list}})\right)_p \\ &= \left((0,\phi_m^{\sigma\ \mathsf{list}}) \vee (f\ \phi_{n-1}^{\sigma\ \mathsf{list}})_c +_c (f\ \phi_{n-1}^{\sigma\ \mathsf{list}})_p\ \phi_{m+1}^{\sigma\ \mathsf{list}}\right)_p \\ &= \phi_m^{\sigma\ \mathsf{list}} \vee ((f\ \phi_{n-1}^{\sigma\ \mathsf{list}})_p\ \phi_{m+1}^{\sigma\ \mathsf{list}})_p\end{aligned}$$

and hence in the end we have

$$\begin{aligned}\tilde{S}(n,m) &= \mathrm{abs}\left((f\ \phi^{\sigma\ \mathsf{list}})_p\ \phi_m^{\sigma\ \mathsf{list}}\right)_p \\ &= \mathrm{abs}\ \phi_m^{\sigma\ \mathsf{list}} \vee \mathrm{abs}((f\ \phi_{n-1}^{\sigma\ \mathsf{list}})_p\ \phi_{m+1}^{\sigma\ \mathsf{list}})_p \\ &= m \vee \tilde{S}(n-1,m+1) \\ &= \tilde{S}(n-1,m+1).\end{aligned}$$

Analysis of cost proceeds in a similar manner. We have again extracted the recurrences we expect from an informal analysis, but instead of those recurrences being in terms of arbitrary values in $W^{\sigma\ \mathsf{list}}$, they are in terms of the length of the argument list.

Stepping back a bit, recall from Section 7.1 that we can apply parametricity to the standard model to reason about the cost of `rev` xs, which seems comparable to what we have just done. But there is a difference. The result from parametricity tells us that the cost of the result is determined by the length of the argument, but it does not tell us how to compute the former in terms of the latter. What we have done here is to formally justify the recurrence that does just that.

7.5 Lower bounds and an application to map fusion

So far, we have focused on extracting recurrences for upper bounds. However, the syntactic bounding theorem is agnostic with respect to the actual interpretation of the size order. We take advantage of this to derive recurrences for upper and lower bounds in the main constructor counting model of Section 7.2. Let us consider the `map` function given in Figure 25. By reasoning that is by now hopefully somewhat mundane, if we set

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\begin{aligned}
&\mathtt{map} = \lambda f^{\rho\to\sigma}, xs^{\rho\ \mathtt{list}}.\\
&\qquad \mathtt{fold}_{\rho\ \mathtt{list}}\ xs\ \mathtt{of}\ \ \mathtt{nil} \Rightarrow \mathtt{nil}\\
&\qquad\qquad\qquad\qquad \mid \mathtt{cons} \Rightarrow (x, r).\mathtt{cons}(f\,x, r)\\
&\mathsf{map} = \lambda (f : \langle\!\langle \rho \rangle\!\rangle \to \|\sigma\|), (xs : \langle\!\langle \rho \rangle\!\rangle\ \mathsf{list})\\
&\qquad \mathsf{fold}_{\langle\!\langle \rho \rangle\!\rangle\ \mathsf{list}}\ xs\ \mathsf{of} \left\{ \begin{array}{l} \mathsf{nil} \Rightarrow (1, \mathsf{nil}) \\ \mid \mathsf{cons} \Rightarrow (x, r).(1 + (f\,x)_c + r_c, \mathsf{cons}((f\,x)_p, r_p)) \end{array} \right\}
\end{aligned}$$

Fig. 25: List map and its extracted recurrence.

$T_{\mathtt{map}\,f}(n) = (\llbracket \mathtt{map} \rrbracket\, f\ n)_c$, then we obtain the recurrence

$$T_{\mathtt{map}\,f}(1) = 1 \qquad T_{\mathtt{map}\,f}(n) = 1 + (f\ \infty)_c + T_{\mathtt{map}\,f}(n-1).$$

Solving this recurrence yields an upper bound of $T_{\mathtt{map}\,f}(n) = n(1 + (f\ \infty)_c)$. Now let us apply this to the two sides of the usual map fusion law

$$\mathtt{map}\,f\ (\mathtt{map}\,g\,xs) = \mathtt{map}\ (f \circ g)\,xs.$$

We hope to show that the right-hand side is less costly than the left. Working through the recurrence extractions, we conclude that the cost of the left-hand side is bounded by $T_{\mathtt{map}\,f\circ\mathtt{map}\,g}(n) = 2n(1 + (g\ \infty)_c + (f\ \infty)_c)$, whereas the right-hand side is bounded by $T_{\mathtt{map}(f\circ g)}(n) = n(1 + (g\ \infty)_c + (f(g\ \infty)_p)_c)$. Even under the assumption that the costs of f and g are independent of their arguments does not result in the desired conclusion, because we only know that these recurrences yield *upper bounds*, and the fact that one upper bound is larger than another tells us nothing about the actual costs. What we would like to know is that these recurrences are tight, and for that we need lower bounds as well.

As we already mentioned, as long as we have a model of the recurrence language in which the interpretation of the size order satisfies the axioms of Figure 11, the bounding theorem holds. So to obtain lower bounds, we would want a model in which the order on the interpretation of C is the reverse of the usual order. That means we would have two models in hand, one that gives us upper bounds and one that gives us lower bounds; we would then have to ensure that the recurrences in each model can be sensibly compared. As it turns out, we can arrange that by using the model in Section 7.2 because the interpretations of the types are all complete upper semi-lattices. We take advantage of the fact that a complete upper semi-lattice is in fact a complete lattice, where greatest lower bounds are defined by $\bigwedge X = \bigvee \{x \mid \forall y \in X : x \leq y\}$. This permits us to define the dual interpretation of the model $(\mathbf{U}_{sm}, \mathbf{U}_{lg}, \{V^\sigma\}_\sigma)$ to be $(\mathbf{U}_{sm}, \mathbf{U}_{lg}, \{(V^*)^\sigma\}_\sigma)$, where $(V^*)^\sigma = (V^\sigma, \leq^*_\sigma)$ and $x \leq^*_\sigma y$ iff $y \leq_\sigma x$. Because all of the size-order axioms except (β_δ) and $(\beta_{\delta\mathsf{fold}})$ are witnessed by identities in $\mathbf{V}$ (i.e., the left- and right-hand sides of the axioms have the same denotation), we can take the semantic functions in $\mathbf{V}^*$ not related to datatypes to be those of $\mathbf{V}$. For datatype-related functions, it is unnecessary to change either $\underline{\mathrm{size}}_F$ or $\underline{\mathsf{C}}_F$; the only change needed is that we define

$$\underline{\mathsf{D}}^*_F(n) = \bigwedge \{a \mid \underline{\mathsf{C}}_F(a) \geq n\}.$$

We can verify that (β_δ) holds by observing that

$$\underline{\mathsf{D}}^*_F(\underline{\mathsf{C}}_F(a)) = \bigwedge \{a' \mid \underline{\mathsf{C}}_F(a') \geq \underline{\mathsf{C}}_F(a)\} \leq a$$

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

and hence $\underline{\mathrm{D}}^*_F(\underline{\mathrm{C}}_F(a)) \geq^* a$ as required. Of course, the value of the destructor is different in this model, but not by much; a routine calculation shows that

$$\underline{\mathrm{D}}_{F_{\sigma\ \mathsf{list}}}(x) = \emptyset \sqcup (\{\bot_\sigma\} \times \downarrow^{\mathbf{N}_1^\infty}(x-1));$$

compare this to the calculation in Section 7.2.

We likewise can define the semantic fold function in this model by

$$\underline{\mathrm{Fold}}^*_F\ s\ x = \bigwedge \{s(\underline{\mathrm{Map}}(\underline{\mathrm{Fold}}\ s)\ z) \mid \underline{\mathrm{C}}_F z \geq x\}$$

Similar to the computation of $\underline{\mathrm{D}}_F$, we have an analogue of Proposition 7: if $f\ n = [\![\mathsf{fold}_{\sigma\ \mathsf{list}}\ y\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons} \Rightarrow (x,r).e_{\mathsf{cons}}\}]\!]\eta\{y \mapsto n\}$, then

$$\begin{aligned} f\ 1 &= \bot_\sigma \\ f\ n &= [\![e_{\mathsf{cons}}]\!]\eta\{x, r \mapsto \bot_\sigma, f(n-1)\}. \end{aligned}$$

Returning to our discussion of comparing the costs of $\mathtt{map}\, f \circ \mathtt{map}\, g$ and $\mathtt{map}(f \circ g)$, we now conclude that $T^{\ell}_{\mathtt{map}\, f \circ \mathtt{map}\, g}(n) = 2n(1 + (g\ \bot)_c + (f\ \bot)_c)$ is a *lower* bound on the cost of $\mathtt{map}\, f \circ \mathtt{map}\, g$, so to show that $\mathtt{map}\ (f \circ g)$ is the more efficient alternative, it suffices to show that

$$n(1 + (g\infty)_c + (f(g\ \infty)_p)_c) \leq_{\mathsf{C}} 2n(1 + (g\ \bot)_c + (f\ \bot)_c),$$

which is trivial when the costs of f and g are independent of their arguments.

8 Recursion

We have not included general recursion in our languages in order to focus on the key idea that different models formally justify various informal cost analyses. The presence of recursion does not change this perspective, but it does complicate the model descriptions in ways orthogonal to our main thrust. We sketch the approach of Kavvos *et al*. (2020) here.

For the syntax, we add recursive definitions to the source language with a standard `letrec` construct and to the recurrence language with a standard fix constructor, corresponding to the usual approach for call-by-value and call-by-name languages. The details are given in Figure 26, where we also give two new size-order rules to replace $(\beta_{\delta\mathsf{fold}})$. In these new rules, $\mathscr{E}$ is an elimination context and $\mathsf{fix}_n\ x.e$ is defined by

$$\mathsf{fix}_0\ x.e = \mathsf{fix}\ x.x \qquad \mathsf{fix}_{n+1}\ x.e = e\{\mathsf{fix}_n\ x.e/x\}.$$

The two rules codify the relation between the size order and the information order that is implicit in the presence of fix: a more defined bound is a better (i.e., smaller) bound. In the presence of nontermination, the bounding relation requires a slight adjustment: $e \preceq E$ provided: if E terminates, then $e\theta \downarrow^n v$, where $n \leq E_c$ and $v \preceq^{\mathrm{val}} E_p$. This is the only place a (standard) operational semantics is needed in the recurrence language, and we are investigating how to eliminate its use.

For the semantics of the recurrence language, we impose additional structure on our applicative structures. We call the new structures *sized domains* and they are defined just like applicative structures, except that for each $U \in \mathbf{U}_{sm}$, $D^U = (D^U, \leq_U, \sqsubseteq_U, \bot_U)$,

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

Source language:

$$\frac{\Gamma, f : \rho \to \rho' \vdash \lambda x.e' : \rho \to \rho' \qquad \Gamma, f : \forall(\rho \to \rho') \vdash e : \sigma}{\Gamma \vdash \texttt{letrec}\ f = \lambda x.e'\ \texttt{in}\ e : \sigma}$$

$$\frac{e\theta\{f \mapsto \lambda x.\texttt{letrec}\ f = \lambda x.e'\ \texttt{in}\ e'\theta\} \downarrow^n v}{\texttt{letrec}\ f = \lambda x.e'\ \texttt{in}\ e\theta \downarrow^{n+1} v}$$

Recurrence language:

$$\frac{\Gamma, x : \sigma \vdash e : \sigma}{\Gamma \vdash \mathsf{fix}\, x.e : \sigma}$$

$$\frac{\{\Gamma \vdash e \leq_\sigma \mathscr{E}[\mathsf{fix}_n\, x.e]\}_{n=0,1,\ldots}}{\Gamma \vdash e \leq_\sigma \mathscr{E}[\mathsf{fix}\, x.e]} \qquad \frac{}{\Gamma \vdash \mathsf{fix}_{n+1}\, x.e \leq_\sigma \mathsf{fix}_n\, x.e}$$

Recurrence extraction:

$$\|\texttt{letrec}\ f = \lambda x.e'\ \texttt{in}\ e\| = \mathsf{let}\ f = \mathsf{fix}\, f.\lambda x.1 +_c \|e'\|\ \mathsf{in}\ \|e\|$$

Fig. 26: Adding general recursion to the source and recurrence languages.

where $(D^U, \leq_U)$ is a preorder as before, and $(D^U, \sqsubseteq_U, \bot_U)$ is a complete partial order. The semantic domains must satisfy two additional constraints:

- If $x \sqsubseteq_U y$, then $y \leq_U x$; and
- If $y_0 \sqsubseteq_U y_1 \sqsubseteq_U \cdots$ and for all i, $x \leq_U y_i$, then $x \leq \bigsqcup y_i$.

That leaves us with verifying that the models that we presented in Section 7 are sized domains. For each of the models, we take $\sqsubseteq_{\mathbf{N}_i^\infty}$ to be the usual flat order with $\bot_{\mathbf{N}_i^\infty} = \infty$ (again, cf. Rosendahl, 1989) extended pointwise and componentwise for functions and products. For sums, set $X \sqsubseteq Y$ if $Y \subseteq X$. It is a straightforward exercise to show that $D^{\rho+\sigma}$ is a CPO that satisfies the constraints just given. To show that we have a model, it suffices to verify that the semantic functions are simultaneously monotone with respect to $\leq$ and continuous with respect to $\sqsubseteq$, after which Proposition 6 can be extended with the clause that $\lambda\!\!\lambda a.[\![\Gamma \vdash e : \sigma]\!]\eta\{x \mapsto a\}$ is continuous with respect to $\sqsubseteq$. Verification of continuity for Case relies on two facts that hold in these models at all types:

- If $a \sqsubseteq a'$ and $b \sqsubseteq b'$, then $(a \vee b) \sqsubseteq (a' \vee b')$; and
- If $a_0 \sqsubseteq a_1 \cdots$ and $b_0 \sqsubseteq b_1 \cdots$, then $\bigsqcup\{a_i \vee b_i\} = (\bigsqcup a_i) \vee (\bigsqcup b_i)$.

Extracting syntactic recurrences from general recursive functions and interpreting them in our models follows the same pattern we have already seen several times. But now the recurrences may have more complex solutions (such as poly-log solutions). For example, Kavvos *et al.* (2020) analyze the standard implementation of merge-sort and interpret it in the model of Section 7.2. Under the usual assumption that the cost of the comparison function is constant the recurrence clause of the semantic recurrence is $T(n) = c + dn + T(n/2)$ for some constants c and d (that arise from the analyses of the functions that divide a list in two and merge two sorted lists), just as expected. Now one may reason in the semantics to establish the $O(n \lg n)$ cost from this recurrence.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

Quick-sort provides an interesting example of how more complex models can be used to capture subtle information that may be necessary for an asymptotic analysis. Quick-sort relies on a partitioning function $\texttt{part} : \alpha \to \alpha\ \texttt{list} \to \alpha\ \texttt{list} \times \alpha\ \texttt{list}$ such that $\texttt{part}\, x\, xs = (ys, zs)$, where ys consists of the elements of xs that are $< x$ and zs those elements that are $\geq x$. A key part of the analysis of quick-sort is the fact that the sum of the lengths of ys and zs is the length of xs. In the models we have presented in Section 7, the extracted recurrence will not yield such a bound. For example, in the main constructor-counting model, the best we can conclude about the extracted recurrence is that in the semantics, $\mathsf{part}\, x\, n = (n, n)$. The problem is that the interpretation of products requires that we choose some specific pair that is a bound on all pairs (k, ℓ) such that $k + \ell = n$, and (n, n) is the least such bound. But we have seen this situation before when it came to interpreting sums, and the solution is the same: instead of taking $V^{\rho \times \sigma} = V^{\rho} \times V^{\sigma}$, we can instead take $V^{\rho \times \sigma} = \mathcal{O}(V^{\rho} \times V^{\sigma})$. While the calculations become more tedious, in such a model we can show that $\mathsf{part}\, x\, n = \{(k, \ell) \mid k + \ell \leq n\}$. However, it turns out this is not quite enough. Both the source and recurrence languages have negative products, which means that projections must be used to extract ys and zs. In the interpretation of the extracted recurrence, projection of a set of pairs maximizes over the corresponding component, and so $\pi_i\, (\mathsf{part}\, x\, n) = n$ (because $n + 0 = 0 + n = n$), which again leads to a weak bound. Instead, we must use positive products with an elimination of the form $\texttt{split}\, (x, y) = e^{\rho \times \sigma}\ \texttt{in}\ e'$. The corresponding elimination form in the recurrence language can be interpreted by maximizing $[\![e']\!]$ over all pairs in $[\![e]\!]$, which is precisely what is needed to carry out the rest of the usual analysis of quick-sort.

9 Related work

We first expand upon a couple of observations that we made earlier and mention some motivating history behind some technical details. Then we address how our work fits into the literature on cost analysis.

We touched on an application of parametricity in Section 7.1. Seidel & Voigtländer (2011) have interpreted free theorems (Wadler, 1989) to obtain relative complexity information. Their work can be viewed as applying parametricity to the standard model, but in a somewhat more general setting of a recurrence language that has a monadic type constructor $\mathsf{C}(\sigma)$ for "complexity of σ," with projections for cost and potential. They define a notion of lifting relations to complexities (much as relations are lifted to inductive types), which allows them to interpret a free theorem such as $f(\texttt{hd}\, xs) = \texttt{hd}(\texttt{map}\, f\, xs)$ in such a way that the interpretations of both sides yield complexity information, and the identity then allows them to conclude, e.g., that the cost of the left-hand side is no greater than that of the right-hand side. With our approach, we would simply extract recurrences from the left- and right-hand sides and reason about them as in Section 7.5. While on the topic of relative cost information, we would be remiss to not mention the type-and-effect system of Çiçek *et al*. (2017), which permits a very precise analysis of the relative cost of different algorithms on the same arguments or the same algorithm on different arguments. We have not investigated whether our techniques can be adapted to provide comparable analyses.

We drew an analogy with abstract interpretation (AI) in Section 7.2.6 and made use of the existence of a Galois connection of the sort that arises in AI in Section 7.4. Rosendahl

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

(1989) uses AI to extract cost bounds directly from a first-order fragment of Lisp. She first defines a program translation similar to our syntactic extraction and interprets it in the standard model D of S-expressions. She then defines an AI from $\mathscr{P}(D)$ into a finite-height lattice of "partial structures," whose values are essentially truncated standard values. Given a notion of size $s : D \to \mathbf{N}$ and a computable bound on $\lambda n.\alpha(\{x \mid s(x) = n\})$, the interpretation of the syntactic recurrence in the abstract domain is a computable upper bound on the cost of the original program. This work is restricted to first-order programs and does not handle branching data structures well (e.g., if $s(t)$ is the number of nodes in the tree t, then for $n > 1$, $\alpha(\{x \mid s(x) = n\})$ is a node structure that is truncated at its children, so the bounds are all trivial). But these ideas may provide an approach to computing bounds on semantic recurrences in models where the semantic recurrence itself is not computable (a situation that does not arise in the models we have presented).

While our notion of potential is drawn most directly from Danner & Royer (2007), it traces back at least to Shultis (1985), who defines a denotational semantics for a simple higher order language that models both the value and the cost of an expression. He develops a system of "tolls," which play a role similar to that of our potentials. The tolls and the semantics are not used directly in calculations, but rather as components in a logic for reasoning about them. Sands (1990) defines a translation scheme in which 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 record information about the future cost of a partially applied function, just as our potentials do. The idea of using denotational semantics to captures cost information has been seen before. We have already mentioned Rosendahl (1989) and Shultis (1985). Van Stone (2003) defines a category-theoretic denotational semantics that uses "cost structures" (these include the $\mathsf{C} \times -$ writer monads we use here) to capture cost information and shows that it is sound with respect to a cost-annotated operational semantics for a higher order language. Our bounding theorem is roughly analogous to Van Stone's soundness theorem, but is a bit more general because we show an inequality (using the size order on the complexity language) instead of an equality, which allows the bounding theorem to apply to models with size abstraction.

Turning now to the literature on cost analysis, constructing resource bounds from source code has a long history in Programming Languages. The earliest work known to the authors is that of Cohen & Zuckerman (1974), which extracts programs that describe costs from an ALGOL60-like language that are intended to be manipulated in an interactive system, and Wegbreit's (1975) METRIC system, which extracts recurrences from simple first-order recursive Lisp programs. An interesting aspect of the latter system is that it is possible to describe probability distributions on the input domain (e.g., the probability that the head of an input list will be some specified value), and the generated bounds incorporate this information. Le Métayer's (1988) ACE system converts FP programs (Backus, 1978) (under a strict operational semantics) to FP programs (under a nonstrict semantics) describing the number of recursive calls of the source program. The first phase is comparable to the cost projection of our recurrence extraction; the potential projection is the original program. Both METRIC and ACE yield nonrecursive upper bounds on the generated cost functions (this is the bulk of the work for ACE). These systems are restricted in their datatypes and compute costs in terms of syntactic values; the notion of "size" is somewhat ad hoc and

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

second class. Many approaches to cost analysis rely on the idea that the cost can be treated as an additional output of the program, or as a piece of program state; Wadler (1992) observed that this can be represented by a monadic translation—though in our case we use the writer monad rather than the state monad, since we do not give programs access to their cost.

There are many approaches to type-based cost analysis (Crary & Weirich, 2000; Hofmann & Jost, 2003; Jost *et al*., 2010; Hoffmann & Hofmann, 2010; Hoffmann *et al*., 2012, 2017; Jost *et al*., 2017; Knoth *et al*., 2019, 2020; Avanzini & Dal Lago, 2017; Çiçek *et al*., 2017; Wang *et al*., 2017; Dal Lago & Gaboardi, 2011; Handley *et al*., 2019; Rajani *et al*., 2021). At a high level, these systems include special-purpose judgments or types that track cost, indexed or refinement types that track the size of values, and a type checking or inference mechanism that can automatically determine some resource bounds. For example, the Automatic Amortized Resource Analysis (AARA) technique of Hoffmann *et al*. (2012), Hoffmann *et al*. (2017), Jost *et al*. (2017), Hofmann & Jost (2003), Jost *et al*. (2010), Hoffmann & Hofmann (2010), with an implementation at Hoffmann (2020), computes cost bounds by introducing a type system with size information that is parameterized by an integer degree, and then performing type inference. If inference is successful, then the program cost can be bounded by a polynomial of at most that degree (and a bound is reported); otherwise it cannot). As its name suggests, AARA automatically incorporates amortization, resulting in tighter bounds for some programs than our extracted recurrences yield (but see Cutler *et al*., 2020 for an extension of our approach to amortized analysis). The basic AARA technique has been extended in numerous ways, e.g., with refinement types (Knoth *et al*., 2019, 2020) for synthesizing programs with desired resource bounds, and for more precise tracking of potential in values. The Timed ML system of Wang *et al*. (2017) also uses refinement types (indexed types in the style of DML Xi & Pfenning, 1999) that permit the user to define datatypes with their own notion of size and to include cost information in the program type. Type inference produces verification conditions that, if solvable, validate the cost information. That cost information may be very concrete, or left more open-ended, in which case the verification conditions end up synthesizing (recurrence) relations that must be satisfied. Avanzini & Dal Lago (2017) develop a non-amortized type-based analysis, which uses a translation similar to our recurrence extraction to explicitly represent the cost as a unary numeral. As a result, the evaluation cost of the original program is reflected in the size of the cost component of the translated program. They then make use of an extension of sized types (Hughes *et al*., 1996) to infer a type for the translated program, which therefore includes a bound on the cost in terms of the size of the arguments.

All of these type-based approaches are impressive in the breadth of successful analyses and/or automation thereof. However, we believe it is nonetheless worth studying cost analysis by recurrence extraction for several reasons. First, the process of inferring bounds using these specialized type systems and their associated solvers is not, in our opinion, very easy for a person to do, while our focus is on formalizing the method that we readily teach students to do. Second, automated approaches necessarily impose some limits on the kinds of bounds that can be inferred and the notions of size that are supported to facilitate inference (though Handley *et al*., 2019 also allows explicit proofs; see the discussion of techniques in proof assistants below). For example, AARA infers polynomial

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

bounds, while our approach (adapted to the setting of general recursion) can produce recurrences with nonpolynomial solutions. Third, type-based approaches make the size and cost an *intrinsic* feature of the code: in approaches based on refinement types, one must, for example, define one tree type where size means number of nodes, and a different tree type where size means height, which causes code duplication if both are necessary; in amortized approaches, one must choose the potential annotations when defining a type (though sometimes this can be mitigated by parametrizing the datatypes Knoth *et al*., 2020). In our approach, cost and size are an *extrinisic* property of the code, so the same function can be interpreted in different models with different notions of size for different analyses, which can be useful, e.g., for a library function that is used in two different programs by other functions that require two different notions of size. That said, this does not address situations where two different notions of size for a type are needed in a single program—one possible solution is a model in which the potential is the pair of these sizes, but this would have similar reuse problems to changing a refinement type to include additional information, in that all existing analyses would formally need to be modified.

Let us now consider work that, like ours, externalizes cost from programs that are typed in a more-or-less standard type system. Avanzini *et al*. (2015) carefully defunctionalize higher order programs to first-order programs in order to take advantage of existing techniques from first-order rewrite systems. This leverages existing technologies to great effect, but does not match the kind of recurrence extraction that we are aiming for in this work. The COSTA project (Albert *et al*., 2012) extracts cost recurrences from Java bytecode; Albert *et al*. (2013) provide techniques for constructing closed forms for both lower and upper bounds on these recurrences. This group has also pushed forward on parallel cost (Albert *et al*., 2018), something that Raymond (2016) has looked into in our setting, but the COSTA work has focused on first-order, low-level languages.

Cutler *et al*. (2020) adapt our technique to handle amortized analysis. Reinforcing our goal of formalizing informal approaches, the source language there includes constructions for describing a credit allocation policy (the banker's method) and extraction of an *amortized* cost recurrence, to which a general theorem applies that total amortized cost bounds total actual cost. The language is sufficient for describing structures like splay trees in which the number of credits allocated to different parts of the structure is not constant, and the source language type system ensures that credits are not misused. The key point is that the amortized cost recurrence is extracted into essentially the same recurrence language as we have presented here, reflecting the fact that the recurrences that we use to describe amortized cost do not themselves refer to credits.

Kavvos *et al*. (2020) give an approach to extending our technique to handle general (as opposed to structural) recursion by using call-by-push-value (CBPV) (Levy, 2003) as an intermediate source language into which both call-by-value and call-by-name can be embedded. While CBPV includes a fine stratification of types into computational and value types, analyzing a program still really just relies on notions of size and cost. Thus, the syntactic recurrence language differs from the one just described only in replacing primitive recursion with a general fixpoint operator, along with corresponding axioms for the size order, thereby changing it from a version of System T with inductive types to a version of PCF with inductive types.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

Atkey (2011), Guéneau *et al*. (2018), Charguéraud & Pottier (2019), and Zhan & Haslbeck (2018) develop imperative program logics for reasoning about cost based on separation logic, essentially by treating the number of timesteps taken as part of the heap. A Coq or Isabelle implementation of these logics allows for reasoning about code, and the subgoals that arise during verification result in synthesizing recurrence relations, which play the role of our syntactic recurrences. While quite sophisticated algorithms and data structures can be analyzed this way, including imperative ones, for analyzing functional programs, we find it more congruous to use (and teach to students) standard functional program verification techniques like inductive reasoning about outputs, as opposed to imperative program verification techniques like weakest precondition/characteristic formula generation. And as we note in Section 10, we conjecture that our approach extends to the analysis of many imperative programs because the description of cost itself is frequently a functional description.

Turning now to semi-automated/manual reasoning in a functional style, Danielsson (2008) verifies a number of lazy functional programs in Agda using a dependent type tracking the number of steps a program takes. McCarthy *et al*. (2018) investigate a variant, implemented in Coq, using a monad parametrized by both the number of steps and a specification, given as a relation between the cost and value. The specifications are used both for functional correctness and for reasoning about cost, and this design allows Coq's extraction to OCaml to erase all costs and reasoning about them. The library also provides a source-to-source translation that translates simply typed code into the monad, inserting appropriate ticks, which is analogous to our recurrence extraction. Radiček *et al*. (2017) define a specification logic for reasoning about monadic costs as an extension of higher order logic.

Benzinger's (2004) ACA system might be the closest in philosophy to ours, in that it extracts (higher order) recurrences from call-by-name NuPRL programs that bound the cost of those programs. There we find (moderately complex) expressions that correspond to applying higher order functions to arguments (necessarily alternating with projections) to describe the cost of a fully applied function argument, corresponding to our notion of higher-order potential. But this does not address more realistic call-by-value or call-by-need evaluation.

Since these approaches (Benzinger, 2004; Danielsson, 2008; Radiček *et al*., 2017; McCarthy *et al*., 2018) take place inside of a general-purpose logic or proof assistant, one can express costs in terms of the sizes of inputs by explicitly referring to an appropriate size function and proving how operations transform the size. Relative to this, a main contribution of our approach is to systematize and partially automate the reasoning about size, in the sense that our semantic interpretation of the potential of a function f gives a direct inductive definition of the fused "size of the result of f on inputs of size –" function. This is possible because we step outside of the programming language into a denotational setting where, e.g., arbitrary maximums exist. We claim that this corresponds better to informal analyses than using the full power of a proof assistant to carefully prove how functions act on sizes, because the fused size-to-size function will simplify in ways that the original function does not. For example, because in these models most or all contexts are monotone in the size order, one can freely ignore branches whose size is dominated by another.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

10 Conclusions and further work

We have presented a technique for extracting cost-and-size recurrences from higher order functional programs that provably bound the operational cost in terms of user-definable notions of size, thereby giving a formal account of the process of many informal cost analyses. The technique applies to the pure fragment of strict languages such as ML and OCaml. Although we have not investigated the question carefully, it also seems that it applies to much reasoning about imperative programs. The reason is that such analysis often consists of extracting *functional* cost recurrences whose validity only depends on the fact that certain imperative operations have certain costs. For example, the analysis of many functions on an arrays depends on the fact that indexed access and update is constant time. But the analysis does not typically result in a recurrence that even refers to an array, much less destructively updates one. In our setting, we would either hard-code the costs of access and update in the syntactic recurrence extraction or we would leave those functions as identifiers and analyze the semantic recurrence under the assumption that those identifiers are interpreted by constant-time functions. The *de facto* standard for such reasoning is Separation Logic, and the work that ours seems closest to in spirit is that of Zhan & Haslbeck (2018). Our goal would be to provide relatively simple approaches to formalizing reasoning about many imperative programs. This is certainly speculative, and we have not investigated how far one can push this idea before requiring the machinery of something comparable to Separation Logic.

A natural direction to extend our work would be to handle cost analysis of lazy languages. Okasaki (1998) describes a technique of amortized analysis in which costs are split into "shared" and "unshared" costs in order to correctly account for the memoization of computations, and we believe our approach can be adapted to formalize this technique. Hackett & Hutton (2019) show that lazy evaluation is a form of "clairvoyant" call-by-value and that cost can be described nondeterministically rather than in terms of shared and unshared costs. We hope to adapt our approach to yield corresponding recurrences, especially as they actually compute costs via an interpretation in a denotational model that appears to mesh nicely with our approach.

We have presented several models making use of different notions of size. It is no surprise that it is easier to work in models with simpler notions of size, and we saw in Section 7.4 that a simpler notion of size corresponds to a more abstract model. Formalizing the connection between more abstract and more concrete models so that information from the latter may be pulled into the former would improve the usefulness of this sort of reasoning. This sounds like an analogy with safety and liveness theorems from abstract interpretation, and this is probably a fruitful direction for further study. More complex models should enable more sophisticated analysis. For example, the average case complexity of deterministic quick-sort can be described by assuming a (uniform) probability distribution on the inputs. That would seem to correspond to interpreting the usually extracted recurrence in a model in which inductive types are interpreted by probability distributions or random variables. Barnaby (2018) has made preliminary progress in this direction, which indicates that it is probably necessary to have at least limited forms of dependent typing in the recurrence language.

We have focused on the extraction of semantic recurrences to show that they are the ones that are expected from informal analysis. We have not studied techniques for solving

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

the semantic recurrences, which in general are higher order functions. Benzinger (2004) discusses techniques for solving them by reducing them to first-order recurrence equations and then using off-the-shelf solvers such as MATHEMATICA and OCRS (Kincaid *et al*., 2017). Another fruitful direction would be to formalize the extracted semantic recurrences in proof assistants and make use of the formalization of standard theorems like the Master Theorem and of asymptotic reasoning as in Guéneau *et al*. (2018). This would permit a formal development in a setting where complete automation is not possible.

The extraction of the syntactic recurrence is straightforward to implement, and a future project is to produce an end-to-end tool from source code to semantic recurrence to solution. We know that automated cost analysis is a complex project that many have attempted, and so this goal as stated is probably too ambitious, and we warn the reader that our thoughts here are pies in the sky at the time of writing. Our vision is more along the lines of an interactive system, in which recurrences are extracted and "easy" ones solved, but allowing the user to step in to provide assertions (hopefully proved!) about the solutions to difficult ones. Familiarity with recurrence extraction as a cost analysis technique would hopefully lower the entry barrier of such a tool. We could also hope that that same familiarity would enable users to work backward from an unexpectedly poor recurrence to the code from which it results (cf. Benzinger, 2004). Wang & Hoffmann (2019) adapt AARA to provide worst-case inputs that validate the tightness of the produced bounds, which could be used to similar effect. Another direction such a project could take would be to pull either the syntactic or the semantic information back as additional interface-level components of a language library, so as to modularize cost reasoning and take advantage of the compositionality of our approach. However, this is not so straightforward. One issue that arises is that the denotation a type that is appropriate for analyzing an algorithm is not necessarily the one that is appropriate for using it. For example, the recurrence extraction approach works best to analyze binary search tree algorithms in terms of their heights, but a client who uses a binary search tree implementation is probably more interested in understanding the cost in terms of the size. This is a setting in which composing recurrences does not work as smoothly as we might hope. Understanding how to mesh them together, and more generally how to hide analyses that possibly require more complex types (such as those by Cutler *et al*., 2020) behind an interface, is ongoing work.

Acknowledgments

Both authors were supported by the National Science Foundation under grant number CCF 1618203. We had a number of helpful discussions about this work with colleagues, including Ed Morehouse, Alex Kavvos, and Joe Cutler. We thank the referees for a number of helpful comments and pointers to literature with which we were initially unfamiliar.

Conflicts of interest

None.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

References

Abadi, M., Cardelli, L., Curien, P.-L. & Lévy, J.-J. (1991) Explicit substitutions. *J. Funct. Program.* **1**(4), 375–416. doi: 10.1017/S0956796800000186.

Aczel, P. (1988) *Non-well-founded Sets*. Center for the Study of Language and Information.

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

Albert, E., Correas, J., Johnsen, E. B., Pun, K. I. & Román-Díez, G. (2018) Parallel cost analysis. *ACM Trans. Comput. Logic* **19**(4), 31:1–31:37. doi: 10.1145/3274278.

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

Atkey, R. (2011) Amortised resource analysis with separation logic. *Logical Methods Comput. Sci.* **7**(2). doi: 10.2168/LMCS-7(2:17)2011.

Avanzini, M. & Dal Lago, U. (2017) Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.* **1**(ICFP), 43:1–43:29. doi: 10.1145/3110287.

Avanzini, M., Dal Lago, U. & Moser, G. (2015) Analyzing the complexity of functional programs: Higher-order mets first-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, Fisher, K. & Reppy, J. (eds), pp. 152–164. doi: 10.1145/2784731.2784753.

Backus, J. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. Assoc. Comput. Mach.* **21**(8), 613–641. doi: 10.1145/359576.359579.

Barnaby, C. (2018) *Denotational Semantics for Probabilistic Recurrences*. Honors thesis, Wesleyan University.

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

Bruce, K., Meyer, A. & Mitchell, J. (1990) The semantics of second-order lambda calculus. *Inf. Comput.* **85**, 76–134. doi: 10.1016/0890-5401(90)90044-I.

Çiçek, E., Barthe, G., Gaboardi, M., Garg, D. & Hoffmann, J. (2017) Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, Castagna, G. & Gordon, A. D. (eds), pp. 316–329. doi: 10.1145/3009837.3009858.

Charguéraud, A. & Pottier, F. (2019) Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *J. Autom. Reasoning* **62**, 331–365. doi: 10.1007/s10817-017-9431-7.

Cohen, J. & Zuckerman, C. (1974) Two languages for estimating program efficiency. *Commun. ACM* **17**(6), 301–308. doi: 10.1145/355616.361015.

Cousot, P. & Cousot, R. (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Graham, R. M. & Harrison, M. A. (eds), pp. 238–252. doi: 10.1145/512950.512973.

Crary, K. & Weirich, S. (2000) Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Wegman, M. & Reps, T. (eds), pp. 184–198. doi: 10.1145/325694.325716.

Cutler, J. W., Licata, D. R. & Danner, N. (2020) Denotational recurrence extraction for amortized analysis. *Proc. ACM Program. Lang.* **4**(ICFP). doi: 10.1145/3408979.

Dal Lago, U. & Gaboardi, M. (2011) Linear dependent types and relative completeness. *Logical Methods Comput. Sci.* **8**(4). doi: 10.2168/LMCS-8(4:11)2012.

Danielsson, N. A. (2008) Lightweight semiformal time complexity analysis for purely functional data structures. In Necula, G. & Wadler, P. (eds), *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 133–144. doi: 10.1145/1328438.1328457.

Danner, N., Licata, D. R. & Ramyaa, R. (2015) Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International*

Conference on Functional Programming, Fisher, K. & Reppy, J. (eds), pp. 140–151. doi: 10.1145/2784731.2784749.

Danner, N., Paykin, J. & Royer, J. S. (2013) A static cost analysis for a higher-order language. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, Might, M. & Horn, D. V. (eds), pp. 25–34. doi: 10.1145/2428116.2428123.

Danner, N. & Royer, J. S. (2007) Adventures in time and space. *Logical Methods Comput. Sci.* **3**(9), 1–53. doi: 10.2168/LMCS-3(1:9)2007.

Davey, B. & Priestley, H. A. (1999) *Introduction to Lattices and Order*. Cambridge University Press.

Fisher, K. & Reppy, J. (eds). (2015) *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*.

Guéneau, A., Charguéraud, A. & Pottier, F. (2018) A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *Programming Languages and Systems: 27th European Symposium on Programming, ESOP 2018*, Ahmed, A. (ed), Lecture Notes in Computer Science, vol. 10801. Springer-Verlag, pp. 533–560. doi: 10.1007/978-3-319-89884-1_19.

Hackett, J. & Hutton, G. (2019) Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.* **3**(ICFP), 114:1–114:23. doi: 10.1145/3341718.

Handley, M. A. T., Vazou, N. & Hutton, G. (2019) Liquidate your assets: Reasoning about resource usage in Liquid Haskell. *Proc. ACM Program. Lang.* **4**(POPL). doi: 10.1145/3371092.

Hoffmann, J. (2020) Resource Aware ML. URL `http://raml.co`.

Hoffmann, J., Aehlig, K. & Hofmann, M. (2012) Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* **34**(3), 14:1–14:62. doi: 10.1145/2362389.2362393.

Hoffmann, J., Das, A. & Weng, S.-C. (2017) Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, Castangna, G. & Gordon, A. D. (eds), pp. 359–373. doi: 10.1145/3009837.3009842.

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

Hofmann, M. & Jost, S. (2003) Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Aiken, A. & Morrisett, G. (eds), pp. 185–197. doi: 10.1145/604131.604148.

Hughes, J., Pareto, L. & Sabry, A. (1996) Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boehm, H. J. & Steele, G. (eds), pp. 410–423. doi: 10.1145/237721.240882.

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

Jost, S., Vasconcelos, P., Florido, M. & Hammond, K. (2017) Type-based cost analysis for lazy functional languages. *J. Autom. Reasoning* **59**(1), 87–120. doi: 10.1007/s10817-016-9398-9.

Kavvos, A., Morehouse, E., Licata, D. R. & Danner, N. (2020) Recurrence extraction for functional programs through call-by-push-value. *Proc. ACM Program. Lang.* **4**(POPL). doi: 10.1145/3371083.

Kincaid, Z., Cyphert, J., Breck, J. & Reps, T. (2017) Non-linear reasoning for invariant synthesis. *Proc. ACM Program. Lang.* **2**(POPL), 54:1–54:33. doi: 10.1145/3158142.

Knoth, T., Wang, D., Polikarpova, N. & Hoffmann, J. (2019) Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pp. 253–268. doi: 10.1145/3314221.3314602.

Knoth, T., Wang, D., Reynolds, A., Hoffmann, J. & Polikarpova, N. (2020) Liquid resource types. *Proc. ACM Program. Lang.* **4**(ICFP). doi: 10.1145/3408988.

Le Métayer, D. (1988) ACE: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.* **10**(2), 248–266. doi: 10.1145/42190.42347.

Levy, P. B. (2003) *Call-by-Push-Value: A Functional-Imperative Synthesis*. Semantic Structures in Computation. Springer-Verlag. doi: 10.1007/978-94-007-0954-6.

McCarthy, J., Fetscher, B., New, M. S., Feltey, D. & Findler, R. B. (2018) A coq library for internal verification of running-times. *Sci. Comput. Program.* **164**, 49–65. doi: 10.1016/j.scico.2017.05.001.

Mitchell, J. C. (1996) *Foundations for Programming Languages*. MIT Press.

Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge University Press. doi: 10.1017/CBO9780511530104.

Radiček, I., Barthe, G., Gaboardi, M., Garg, D. & Zuleger, F. (2017) Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.* **2**(POPL). doi: 10.1145/3158124.

Rajani, V., Gaboardi, M., Garg, D. & Hoffmann, J. (2021) A unifying type-theory for higher-order (amortized) cost analysis. *Proc. ACM Program. Lang.* **5**(POPL). doi: 10.1145/3434308.

Raymond, J. (2016) *Extracting Cost Recurrences from Sequential and Parallel Functional Programs*. M.A. thesis, Wesleyan University.

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

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

Seidel, D. & Voigtländer, J. (2011) Improvements for free. In *Proceedings of the 9th Workshop on Quantitative Aspects of Programming Languages (QAPL 2011)*, Massink, M. & Norman, G. (eds), vol. 57, pp. 89–103. doi: 10.4204/eptcs.57.7.

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

Smyth, M. & Plotkin, G. (1982) The category-theoretic solution of recursive domain equations. *SIAM J. Comput.* **11**(4), 761–783. doi: 10.1137/0211062.

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

Wadler, P. (1989) Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming Lanuages and Computer Architecture*, Stoy, J. E. (ed), pp. 347–359. doi: 10.1145/99370.99404.

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

Wang, D. & Hoffmann, J. (2019) Type-guided worst-case input generation. *Proc. ACM Program. Lang.* **3**(POPL), 13:1–13:30. doi: 10.1145/3290326.

Wang, P., Wang, D. & Chlipala, A. (2017) Timl: A functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.* **1**(OOPSLA). doi: 10.1145/3133903.

Wegbreit, B. (1975) Mechanical program analysis. *Commun. Assoc. Comput. Mach.* **18**(9), 528–539. doi: 10.1145/361002.361016.

Xi, H. & Pfenning, F. (1999) Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 214–227. doi: 10.1145/292540.292560.

Zhan, B. & Haslbeck, M. P. L. (2018) Verifying asymptotic time complexity of imperative programs in Isabelle. In *International Joint Conference on Automated Reasoning, IJCAR 2018*, Galmiche, D., Schulz, S. & Sebastiani, R. (eds), Lecture Notes in Computer Science, vol. 10900. Springer International Publishing, pp. 532–548. doi: 10.1007/978-3-319-94205-6_35.

1 Type preservation for the source language

Type preservation depends on the usual substitution lemmas.

Lemma 10. *If* $\Gamma, x : \rho \vdash e(\theta - x) : \sigma$ *and* $\vdash v : \rho$, *then* $\Gamma \vdash e\theta\{x \mapsto v\} : \sigma$.

Lemma 11. *If* $y : \rho \vdash v' : \sigma$ *and* $\vdash v : \rho$, *then* $v'\{v/y\}$ *is a value and* $\vdash v'\{v/y\} : \sigma$.

We now have the type preservation theorem.

Theorem (Type preservation, Theorem 1)**.**

1. *If* $\_ \vdash e\theta : \sigma$ *and* $e\theta \downarrow v$, *then* $\vdash v : \sigma$.
2. *If* $\_ \vdash (\mathtt{map}_F\, y.v'' \,\mathtt{into}\, v')\theta : F[\sigma]$ *and* $\mathtt{map}_F\, y.v'' \,\mathtt{into}\, v' \downarrow v$, *then* $\vdash v : F[\sigma]$.
3. *If* $\_ \vdash \mathtt{mapv}_F\, y.v'' \,\mathtt{into}\, v' : F[\sigma]$ *and* $\mathtt{mapv}_F\, y.v'' \,\mathtt{into}\, v' \downarrow v$, *then* $\vdash v : F[\sigma]$.

Proof The proof is a simultaneous induction on the height of the derivation that referred to in each part. We give just a few of the more interesting cases, starting with part (1).

CASE: $x\theta \downarrow \theta(x)$. By the hypothesis, $\_ \vdash x\theta : \sigma$, so by the typing rules for closures, there must be some Γ' such that $\Gamma'(x) = \forall\vec{\alpha}.\rho$ and $\sigma = \rho\{\vec{\sigma}/\vec{\alpha}\}$, and θ is a Γ'-environment. But that means that in particular, $\vdash \theta(x) : \rho\{\vec{\sigma}/\vec{\alpha}\}$, as required.

CASE: $(\mathtt{case}\, e\, \mathtt{of}\, \{x.e_i\}_{i=0,1})\theta \downarrow v$. The typing must have the form

$$\dfrac{\dfrac{\Gamma \vdash e : \sigma_0 + \sigma_1 \qquad \{\Gamma, x : \sigma_i \vdash e_i : \sigma\}_{i=0,1}}{\Gamma \vdash \mathtt{case}\, e\, \mathtt{of}\, \{x.e_i\}_{i=0,1} : \sigma} \qquad \theta \text{ a } \Gamma\text{-environment}}{\_ \vdash (\mathtt{case}\, e\, \mathtt{of}\, \{x.e_i\}_{i=0,1})\theta : \sigma}$$

and the evaluation must have the form

$$\dfrac{e\theta \downarrow \iota_i\,(v_i) \qquad e_i\theta\{x \mapsto v_i\} \downarrow v}{(\mathtt{case}\, e\, \mathtt{of}\, \{x.e_i\}_{i=0,1})\theta \downarrow v}$$

By definition $\_ \vdash e\theta : \sigma_0 + \sigma_1$, so by the induction hypothesis, $\vdash \iota_i\, v_i : \sigma_0 + \sigma_1$, and hence by inversion, $\vdash v_i : \sigma_i$. That means that $\theta\{x \mapsto v_i\}$ is a $(\Gamma, x : \sigma_i)$-environment, and hence $\_ \vdash e_i\theta\{x \mapsto v_i\} : \sigma$. So by the induction hypothesis, $\vdash v : \sigma$, as required.

CASE: $(\lambda x.e)\theta \downarrow (\lambda x.e)\theta$. If $\_ \vdash (\lambda x.e)\theta : \sigma \to \sigma'$, then we must show that $\_ \vdash (\lambda x.e)\theta : \sigma \to \sigma'$ as a value. For this we must show that $\_ \vdash (\lambda x.e)\theta : \sigma \to \sigma'$ as a closure, which is precisely the hypothesis we started with.

CASE: $(e_0\, e_1)\theta \downarrow v$. The typing has the form

$$\dfrac{\dfrac{\Gamma \vdash e_0 : \rho \to \sigma \qquad \Gamma \vdash e_1 : \rho}{\Gamma \vdash e_0\, e_1 : \sigma} \qquad \theta \text{ a } \Gamma\text{-environment}}{\_ \vdash (e_0\, e_1)\theta : \sigma}$$

and the evaluation has the form

$$\dfrac{e_0\theta \downarrow (\lambda x.e_0')\theta_0' \qquad e_1\theta \downarrow v_1 \qquad e_0'\theta_0'\{x \mapsto v_1\} \downarrow v}{(e_0\, e_1)\theta \downarrow v}$$

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

Since θ is a Γ-environment, $\_ \vdash e_0\theta : \rho \to \sigma$, so by the induction hypothesis, $\vdash (\lambda x.e_0')\theta_0' : \rho \to \sigma$ and similarly $\vdash v_1 : \rho$. By definition we have that there is some Γ' such that $\Gamma' \vdash \lambda x.e_0' : \rho \to \sigma$ and θ_0' is a Γ'-environment; by inversion we have that $\Gamma', x : \rho \vdash e_0' : \sigma$. Since $\vdash v_1 : \rho$, $\theta_0'\{x \mapsto v_1\}$ is a $(\Gamma', x : \rho)$-environment, and so $\_ \vdash e_0\theta_0'\{x \mapsto v_1\} : \sigma$, and so by the induction hypothesis, $\vdash v : \sigma$, as required.

CASE: $(\mathtt{fold}_\delta\ e'\ \mathtt{of}\ x.e)\theta \downarrow v$. The typing must have the form

$$\dfrac{\dfrac{\Gamma \vdash e' : \delta \qquad \Gamma, x : F[\sigma\ \mathtt{susp}] \vdash e : \sigma}{\Gamma \vdash \mathtt{fold}_\delta\ e'\ \mathtt{of}\ x.e : \sigma} \qquad \theta \text{ a } \Gamma\text{-environment}}{\_ \vdash (\mathtt{fold}_\delta\ e'\ \mathtt{of}\ x.e)\theta : \sigma}$$

and the evaluation must have the form

$$\dfrac{e'\theta \downarrow \mathtt{c}_\delta\ v' \qquad \mathtt{mapv}_F\ y.(\mathtt{delay}\ (\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e))\theta\ \mathtt{into}\ v' \downarrow v'' \qquad e\theta\{x \mapsto v''\} \downarrow v}{(\mathtt{fold}_\delta\ e'\ \mathtt{of}\ x.e)\theta \downarrow v}$$

where without loss of generality we assume $y \notin \operatorname{dom}\Gamma$ and $y \notin \operatorname{dom}\theta$. By the assumptions and induction hypothesis, $\vdash \mathtt{c}_\delta\ v' : \delta$, and so by inversion, $\vdash v' : F[\delta]$. From $\Gamma, x : F[\sigma\ \mathtt{susp}] \vdash e : \sigma$, we conclude that $\Gamma, y : \delta \vdash \mathtt{delay}\ (\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e) : \sigma\ \mathtt{susp}$. Since θ is a Γ-environment, we conclude that $y : \delta \vdash (\mathtt{delay}\ (\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e))\theta : \sigma\ \mathtt{susp}$. These two judgments allow us to conclude that $\_ \vdash \mathtt{mapv}_F\ y.(\mathtt{delay}\ (\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e))\theta\ \mathtt{into}\ v' : F[\sigma\ \mathtt{susp}]$, so by the induction hypothesis applied to the evaluation of the mapv expression, $\vdash v'' : F[\sigma\ \mathtt{susp}]$. That means that $\theta\{x \mapsto v''\}$ is a $(\Gamma, x : F[\sigma\ \mathtt{susp}])$-environment, and so by the induction hypothesis applied to the evaluation of $e\theta\{x \mapsto v''\}$, $\vdash v : \sigma$, as required.

For (2), suppose $(\mathtt{map}_F\ y.v'\ \mathtt{into}\ e)\theta \downarrow v$. The typing must have the form

$$\dfrac{\dfrac{y : \rho \vdash v' : \sigma \qquad \Gamma \vdash e : F[\rho]}{\Gamma \vdash \mathtt{map}_F\ y.v'\ \mathtt{into}\ e : F[\sigma]} \qquad \theta \text{ a } \Gamma\text{-environment}}{\_ \vdash (\mathtt{map}_F\ y.v'\ \mathtt{into}\ e)\theta : F[\sigma]}$$

and the evaluation the form

$$\dfrac{e\theta \downarrow^n v'' \qquad \mathtt{mapv}_F\ y.v'\ \mathtt{into}\ v'' \downarrow v}{(\mathtt{map}_F\ y.v'\ \mathtt{into}\ e)\theta \downarrow^n v}$$

As in previous cases, $\vdash v'' : F[\rho]$, and so $\_ \vdash \mathtt{mapv}_F\ y.v'\ \mathtt{into}\ v'' : F[\sigma]$, and so the result follows from the induction hypothesis applied to this mapv expression.

We now prove (3).

CASE: $\mathtt{mapv}_t\ y.v'\ \mathtt{into}\ v \downarrow v'\{v/y\}$. From the typing assumption, we have that $y : \rho \vdash v' : \sigma$ and $\vdash v : \rho$, so the result follows from Lemma 11.

CASE: $\mathtt{mapv}_{\rho \to F}\ y.v'\ \mathtt{into}\ (\lambda x.e)\theta \downarrow (\lambda x.\mathtt{map}_F\ y.v'\ \mathtt{into}\ e)\theta$. The typing must have the form

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\dfrac{\dfrac{\dfrac{\Gamma, x:\rho \vdash e : F[\rho']}{\Gamma \vdash \lambda x.e : \rho \to F[\rho']} \qquad \theta \text{ a } \Gamma\text{-environment}}{\vdash (\lambda x.e)\theta : \rho \to F[\rho']} \qquad y:\rho \vdash v' : \sigma}{\_ \vdash \mathtt{mapv}_{\rho \to F}\, y.v' \ \mathtt{into}\ (\lambda x.e)\theta : \rho \to F[\sigma']}$$

Thus, we obtain a typing of the value as

$$\dfrac{\dfrac{\dfrac{\dfrac{y:\rho' \vdash v' : \sigma \qquad \Gamma, x:\rho \vdash e : F[\rho']}{\Gamma, x:\rho \vdash \mathtt{map}_F\, y.v' \ \mathtt{into}\ e : F[\sigma]}}{\Gamma \vdash \lambda x.\mathtt{map}_F\, y.v' \ \mathtt{into}\ e : \rho \to F[\sigma]} \qquad \theta \text{ a } \Gamma\text{-environment}}{\_ \vdash (\lambda x.\mathtt{map}_F\, y.v' \ \mathtt{into}\ e)\theta : \rho \to F[\sigma]}$$

■

2 Typeability of extracted recurrences

In this appendix we prove that extracted recurrences are typeable. It is worth remembering that $\langle\!\langle \rho \to \sigma \rangle\!\rangle = \langle\!\langle \rho \rangle\!\rangle \to \|\sigma\|$, so extraction "commutes" with type substitution in the expected way.

Lemma 12. $\langle\!\langle \rho\{\vec{\sigma}/\vec{\alpha}\} \rangle\!\rangle = \langle\!\langle \rho \rangle\!\rangle\{\langle\!\langle \vec{\sigma} \rangle\!\rangle/\vec{\alpha}\}$. *Since shape functors are a subset of types, this implies that* $\langle\!\langle F\{\vec{\sigma}/\vec{\alpha}\} \rangle\!\rangle = \langle\!\langle F \rangle\!\rangle\{\langle\!\langle \vec{\sigma} \rangle\!\rangle/\vec{\alpha}\}$ *and* $\langle\!\langle F[\rho] \rangle\!\rangle = \langle\!\langle F \rangle\!\rangle[\langle\!\langle \rho \rangle\!\rangle]$

Lemma 13. *If* $\Gamma, x:\tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$, *then* $\Gamma \vdash e\{e'/x\} : \tau$.

Lemma 14. *If* $\Gamma \vdash c : \mathsf{C}$ *and* $\Gamma \vdash e : \|\sigma\|$, *then* $\Gamma \vdash c +_c e : \|\sigma\|$.

Proposition (Typeability of extracted recurrences, Prop. 1)**.** *If* $\Gamma \vdash e : \sigma$ *is in the core language, then* $\langle\!\langle \Gamma \rangle\!\rangle \vdash \|e\| : \|\sigma\|$.

Proof The proof is by induction on the derivation of $\Gamma \vdash e : \sigma$; we just do a few of the cases, since they are all fairly routine.

CASE: $\Gamma, x:\forall\vec{\alpha}.\rho \vdash x : \rho\{\vec{\sigma}/\vec{\alpha}\}$. $\|x\| = (0, x,)$ and $\|\rho\{\vec{\sigma}/\vec{\alpha}\}\| = \mathsf{C} \times \langle\!\langle \rho\{\vec{\sigma}/\vec{\alpha}\} \rangle\!\rangle = \mathsf{C} \times \langle\!\langle \rho \rangle\!\rangle\{\langle\!\langle \vec{\sigma} \rangle\!\rangle/\vec{\alpha}\}$; the recurrence language typing is

$$\dfrac{\dfrac{}{\langle\!\langle \Gamma \rangle\!\rangle, x:\forall\vec{\alpha}.\langle\!\langle \rho \rangle\!\rangle \vdash 0 : \mathsf{C}} \qquad \dfrac{}{\langle\!\langle \Gamma \rangle\!\rangle, x:\forall\vec{\alpha}.\langle\!\langle \rho \rangle\!\rangle \vdash x : \forall\vec{\alpha}.\langle\!\langle \rho \rangle\!\rangle}}{\langle\!\langle \Gamma \rangle\!\rangle, x:\forall\vec{\alpha}.\langle\!\langle \rho \rangle\!\rangle \vdash x : \langle\!\langle \rho \rangle\!\rangle\{\langle\!\langle \vec{\sigma} \rangle\!\rangle/\vec{\alpha}\}}$$
$$\langle\!\langle \Gamma \rangle\!\rangle, x:\forall\vec{\alpha}.\langle\!\langle \rho \rangle\!\rangle \vdash (0, x) : \mathsf{C} \times \langle\!\langle \rho \rangle\!\rangle\{\langle\!\langle \vec{\sigma} \rangle\!\rangle/\vec{\alpha}\}$$

CASE: $\Gamma \vdash \lambda x.e : \rho \to \sigma$. $\|\rho \to \sigma\| = \mathsf{C} \times (\langle\!\langle \rho \rangle\!\rangle \to \|\sigma\|)$ and we have $\Gamma, x:\rho \vdash e : \sigma$, so by the induction hypothesis, $\langle\!\langle \Gamma \rangle\!\rangle, x:\langle\!\langle \rho \rangle\!\rangle \vdash \|e\| : \|\sigma\|$ and hence

$$\dfrac{\Gamma \vdash 0 : \mathsf{C} \qquad \dfrac{\langle\!\langle \Gamma \rangle\!\rangle, x:\langle\!\langle \rho \rangle\!\rangle \vdash \|e\| : \|\sigma\|}{\Gamma \vdash \lambda(x:\langle\!\langle \rho \rangle\!\rangle).\|e\| : \langle\!\langle \rho \rangle\!\rangle \to \|\sigma\|}}{\langle\!\langle \Gamma \rangle\!\rangle \vdash (0, \lambda(x:\langle\!\langle \rho \rangle\!\rangle).\|e\|) : \mathsf{C} \times (\langle\!\langle \rho \rangle\!\rangle \to \|\sigma\|)}$$

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

CASE: $\Gamma \vdash \texttt{fold}_\delta\ e'\ \texttt{of}\ x.e : \sigma$. The typing derivation has the form

$$\frac{\Gamma \vdash e' : \delta \qquad \Gamma, x : F[\sigma\ \texttt{susp}] \vdash e : \sigma}{\Gamma \vdash \texttt{fold}_\delta\ e'\ \texttt{of}\ x.e : \sigma}$$

so by the induction hypothesis $\langle\!\langle \Gamma \rangle\!\rangle \vdash \|e'\| : \mathsf{C} \times \langle\!\langle \delta \rangle\!\rangle$ and $\langle\!\langle \Gamma \rangle\!\rangle, x : \langle\!\langle F \rangle\!\rangle[\|\sigma\|] \vdash \|e\| : \|\sigma\|$. Writing (c', p') for $\|e'\|$, by inversion we have that $\langle\!\langle \Gamma \rangle\!\rangle \vdash c' : \mathsf{C}$ and $\langle\!\langle \Gamma \rangle\!\rangle \vdash p' : \langle\!\langle \delta \rangle\!\rangle$. We must show that $\langle\!\langle \Gamma \rangle\!\rangle \vdash c' +_c \mathsf{fold}_{\langle\!\langle \delta \rangle\!\rangle}\ p'\ \mathsf{of}\ (x : \langle\!\langle F \rangle\!\rangle[\|\sigma\|]).1 +_c \|e\| : \|\sigma\|$. This follows directly from the typings given by the induction hypothesis, making use of the fact that $\langle\!\langle F\{\eta\} \rangle\!\rangle = \langle\!\langle F \rangle\!\rangle\{\langle\!\langle \vec{\sigma} \rangle\!\rangle/\vec{\alpha}\}$ and Lemma 14.

CASE: $\Gamma \vdash \texttt{let}\ x = e'\ \texttt{in}\ e : \sigma$. The typing derivation has the form

$$\frac{\Gamma \vdash e' : \rho \qquad \Gamma, x : \forall\vec{\alpha}.\rho \vdash e : \sigma \qquad \vec{\alpha} \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash \texttt{let}\ x = e'\ \texttt{in}\ e : \sigma}$$

The induction hypothesis tells us that $\langle\!\langle \Gamma \rangle\!\rangle \vdash \|e'\| : \|\rho\|$, so if $\|e\|' = (c', p')$, then $\langle\!\langle \Gamma \rangle\!\rangle \vdash c' : \mathsf{C}$ and $\langle\!\langle \Gamma \rangle\!\rangle \vdash p' : \langle\!\langle \rho \rangle\!\rangle$. From the latter we conclude that $\langle\!\langle \Gamma \rangle\!\rangle \vdash \Lambda\vec{\alpha}.p' : \forall\vec{\alpha}.\langle\!\langle \rho \rangle\!\rangle$ because $\vec{\alpha} \notin \mathrm{ftv}(\Gamma)$ implies that $\vec{\alpha} \notin \mathrm{ftv}(\langle\!\langle \Gamma \rangle\!\rangle)$. The induction hypothesis also tells us that $\langle\!\langle \Gamma \rangle\!\rangle, x : \forall\vec{\alpha}.\langle\!\langle \rho \rangle\!\rangle \vdash \|e\| : \|\sigma\|$. Together with Lemma 13 we conclude that $\langle\!\langle \Gamma \rangle\!\rangle \vdash \|e\|\{\Lambda\vec{\alpha}.p'/x\} : \|\sigma\|$ and so Lemma 14 yields the desired conclusion.

■

3 The syntactic bounding theorem

In this appendix, we prove the syntactic bounding theorem (Theorem 5). The proof relies on two lemmas that describe bounding for `mapv` and `fold` expressions.

Lemma 15 (Syntactic bounding for `mapv`)**.** *Suppose* $\mathrm{ftv}(F) \subseteq \{t\}$ *and that the following all hold:*

1. $y : \rho \vdash v' : \sigma$ *and* $y : \langle\!\langle \rho \rangle\!\rangle \vdash E' : \langle\!\langle \sigma \rangle\!\rangle$.
2. $\_ \vdash v : F[\rho]$ *and* $\mathscr{E} :: v \preceq^{\mathrm{val}}_{F[\rho]} E$;
3. *If* $\_ \vdash w_0 : \rho$ *and* $\mathscr{E}_0 :: w_0 \preceq^{\mathrm{val}}_{\rho} E_0$ *is a subderivation of* $\mathscr{E}$ *then* $v'\{w_0/y\} \preceq^{\mathrm{val}}_{\sigma} E'\{E_0/y\}$;
4. $\mathtt{mapv}_F\ y.v'\ \mathtt{into}\ v \downarrow v''$.

Then $v'' \preceq^{\mathrm{val}}_{F[\sigma]} (\langle\!\langle F \rangle\!\rangle)[(y : \langle\!\langle \rho \rangle\!\rangle).E', E]$.

Proof The proof is by induction on F.

CASE: $F = t$. Assumption (4) tells us that $v'' = v'\{v/y\}$, so we must show that $v'\{v/y\} \preceq^{\mathrm{val}}_{\sigma} E'\{E/y\}$, which follows from assumption (3), taking v_0 and E_0 to be v and E, respectively.

CASE: $F = \tau_0$. Assumption (4) tells us that $v'' = v$, so we must show that $v \preceq^{\mathrm{val}}_{\tau_0} E$, which follows from assumption (2).

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

CASE: $F = F_0 \times F_1$. Assumption (2) and inversion tells us that $v = (v_0, v_1)$, and assumption (4) tells us that $v'' = (v_0'', v_1'')$, where

$$\frac{\{\mathtt{mapv}_{F_i}\, y.v' \;\mathtt{into}\; v_i \downarrow v_i''\}_{i=0,1}}{\mathtt{mapv}_{F_0 \times F_1}\, y.v' \;\mathtt{into}\; (v_0, v_1) \downarrow (v_0'', v_1'')}$$

We must show that $(v_0'', v_1'') \preceq^{\mathrm{val}} (\langle\!\langle F_0 \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', \pi_0\, E], \langle\!\langle F_1 \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', \pi_1\, E])$, for which it suffices to show that $v_i'' \preceq^{\mathrm{val}} \langle\!\langle F_i \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', \pi_i\, E]$ for $i = 0, 1$. To do so, we apply the induction hypothesis taking F_i for F, v_i for v, $\pi_i\, E$ for E, and v_i'' for v''. Verifying the assumptions is straightforward, noting that (3) follows because the derivation that $v_i \preceq^{\mathrm{val}} \pi_i\, E$ is a subderivation of $(v_0, v_1) \preceq^{\mathrm{val}} E$.

CASE: $F = F_0 + F_1$. Assumption (2) and inversion tells us that $v = \iota_i\, v_i$, where there is E_i such that $v_i \preceq^{\mathrm{val}} E_i$ and $\iota_i\, E_i \leq_{\langle\!\langle F_i \rangle\!\rangle[\rho]} E$. Assumption (4) tells us that $v'' = \iota_i\, v_i''$, where

$$\frac{\mathtt{mapv}_{F_i}\, y.v' \;\mathtt{into}\; v_i \downarrow v_i''}{\mathtt{mapv}_{F_0 + F_1}\, y.v' \;\mathtt{into}\; \iota_i\, v_i \downarrow \iota_i\, v_i''}$$

We must show that

$$\iota_i\, v_i'' \preceq^{\mathrm{val}} \mathsf{case}\; E \;\mathsf{of}\; \{(x : \langle\!\langle F_i \rangle\!\rangle[\langle\!\langle \rho \rangle\!\rangle]).\iota_i\, (\langle\!\langle F_i \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', x]\}_{i=0,1}.$$

Let us write E^* for the right-hand side. Now we must show that there is E_i'' such that $v_i'' \preceq^{\mathrm{val}} E_i''$ and $\iota_i\, E_i'' \leq E^*$. We apply the induction hypothesis taking F_i for F, v_i for v, E_i for E, and v_i'' for v'' to conclude that $v_i'' \preceq^{\mathrm{val}} E_i''$ where $E_i'' = \langle\!\langle F_i \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', E_i]$, and we notice that

$$\begin{aligned}
\iota_i\, E_i'' &= \iota_i\, (\langle\!\langle F_i \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', E_i]) \\
&\leq \mathsf{case}\; \iota_i\, E_i \;\mathsf{of}\; \{(x : \langle\!\langle F_i \rangle\!\rangle[\langle\!\langle \rho \rangle\!\rangle]).\iota_i\, (\langle\!\langle F_i \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', x]\}_{i=0,1} \\
&\leq E^*
\end{aligned}$$

as required. The assumptions for the induction hypothesis are straightforward to verify, noting that (3) follows because the derivation that $v_i \preceq^{\mathrm{val}} E_i$ is a subderivation of $\iota_i\, v_i \preceq^{\mathrm{val}} E$.

CASE: $F = \tau_0 \to F_0$. Assumption (2) and inversion tells us that $v = (\lambda x.e)\theta$, and assumption (4) tells us that $v'' = (\lambda x.\mathtt{map}_F\, y.v' \;\mathtt{into}\; e)\theta$. We must show that

$$\begin{aligned}
(\lambda x.\mathtt{map}_F\, y.v' \;\mathtt{into}\; e)\theta &\preceq^{\mathrm{val}} \langle\!\langle \tau_0 \to F_0 \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', E] \\
&= \lambda(x : \langle\!\langle \tau_0 \rangle\!\rangle).((E\, x)_c, \langle\!\langle F_0 \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', (E\, x)_p]).
\end{aligned}$$

To do so, fix $v_1 \preceq^{\mathrm{val}}_{\tau_0} E_1$; it suffices to show that $(\mathtt{map}_F\, y.v' \;\mathtt{into}\; e)\theta\{x \mapsto v_1\} \preceq_{F[\rho]} ((E\, E_1)_c, \langle\!\langle F \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', (E\, E_1)_p])$. The evaluation of the left-hand side has the form

$$\frac{e\theta\{x \mapsto v_1\} \downarrow^n w' \qquad \mathtt{mapv}_F\, y.v' \;\mathtt{into}\; w' \downarrow w}{(\mathtt{map}_F\, y.v' \;\mathtt{into}\; e)\theta\{x \mapsto v_1\} \downarrow^n w}$$

so by Lemma 4 it suffices to show that $n \leq (E\, E_1)_c$ and $w \preceq^{\mathrm{val}} \langle\!\langle F \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', (E\, E_1)_p]$. Recalling that $v = (\lambda x.e)\theta$ and $v \preceq^{\mathrm{val}} E$ by assumption (2), we have that $e\theta\{x \mapsto v_1\} \preceq E\, E_1$, and hence $n \leq (E\, E_1)_c$ (our first obligation) and $w' \preceq^{\mathrm{val}} (E\, E_1)_p$.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

To show that $w \preceq^{\mathrm{val}} \langle\!\langle F \rangle\!\rangle[(y : \langle\!\langle \rho \rangle\!\rangle).E', (E\,E_1)_p]$ we apply the induction hypothesis taking F_0 for F, w' for v, $(E\,E_1)_p$ for E, and w for v''. Assumptions (1), (2), and (4) are straightforward to verify. For assumption (3), suppose that $\_ \vdash w_0 : \rho$ and $\mathscr{E}_0 :: w_0 \preceq^{\mathrm{val}} E_0$ is a subderivation of $\mathscr{E}' :: w' \preceq^{\mathrm{val}} (E\,E_1)_p$. We need to show that $v'\{w_0/y\} \preceq^{\mathrm{val}} E'\{E_0/y\}$. To do so, it suffices to show that $\mathscr{E}_0$ is a subderivation of $\mathscr{E} :: v \preceq^{\mathrm{val}} E$, and for this it suffices to show that $\mathscr{E}'$ is a subderivation of $\mathscr{E}$. This follows from examining $\mathscr{E}$:

$$\dfrac{\cdots \qquad \dfrac{e\theta\{x \mapsto v_1\} \downarrow^n w' \qquad n \preceq (E\,E_1)_c \qquad \overset{\mathscr{E}'}{w' \preceq^{\mathrm{val}} (E\,E_1)_p}}{e\theta\{x \mapsto v_1\} \preceq E\,E_1} \qquad \cdots}{(\lambda x.e)\theta \preceq^{\mathrm{val}} E}$$

■

Lemma 16 (Syntactic bounding for `fold`)**.** *Suppose the following all hold:*

1. $(\Gamma, x : F[\sigma\ \mathtt{susp}] \vdash e : \sigma) \preceq_\sigma (\langle\!\langle \Gamma \rangle\!\rangle, x : \langle\!\langle F[\sigma\ \mathtt{susp}] \rangle\!\rangle \vdash E : \|\sigma\|)$*;*
2. $\theta \preceq^{\mathrm{val}}_{\Gamma - x} \Theta$ *(w.l.o.g., $x \notin \mathrm{dom}\,\Theta$);*
3. $v' \preceq^{\mathrm{val}}_\delta E'$.

Then $(\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e)\theta\{y \mapsto v'\} \preceq_\sigma \mathsf{fold}_{\langle\!\langle\delta\rangle\!\rangle}\ E'\ \mathsf{of}\ (x : \langle\!\langle F \rangle\!\rangle[\|\sigma\|]).1 +_c E\{\Theta\}$.

Proof The proof is by induction on the derivation of assumption (3), which necessarily ends with the rule

$$\dfrac{v' \preceq^{\mathrm{val}}_{F,\delta} E'' \qquad \mathsf{c}_{\langle\!\langle\delta\rangle\!\rangle}\ E'' \preceq_{\langle\!\langle\delta\rangle\!\rangle} E'}{\mathsf{c}_\delta\ v' \preceq^{\mathrm{val}}_\delta E'}$$

To reduce notational clutter, we will write $E^*[z]$ for $\mathsf{fold}_{\langle\!\langle\delta\rangle\!\rangle}\ z\ \mathsf{of}\ (x : \langle\!\langle F \rangle\!\rangle[\|\sigma\|]).1 +_c E\{\Theta\}$, so we must show that $(\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e)\theta\{y \mapsto \mathsf{c}_\delta\ v'\} \preceq E^*[E']$. Using the axioms for $\preceq$, we have that $E^*[E'] \geq E^*[\mathsf{c}_{\langle\!\langle\delta\rangle\!\rangle}\ E''] \geq 1 +_c E\{\Theta\}\{\langle\!\langle F \rangle\!\rangle[(y : \langle\!\langle\delta\rangle\!\rangle).E^*[y], E'']/x\}$. The evaluation of interest has the form

$$\dfrac{y\theta\{y \mapsto \mathsf{c}_\delta\ v'\} \downarrow^0 \mathsf{c}_\delta\ v' \qquad \mathtt{mapv}_F\ y.(\mathtt{delay}\,(\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e))\theta\ \mathtt{into}\ v' \downarrow v'' \qquad e\theta\{x \mapsto v''\} \downarrow^n v}{(\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e)\theta\{y \mapsto \mathsf{c}_\delta\ v'\} \downarrow^{n+1} v}$$

We apply Lemma 15 by taking F for F, δ for ρ, $(\mathtt{delay}\,(\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e))\theta$ for v', $E^*[y]$ for E', v' for v, E'' for E, and v'' for v'' (we verify the assumptions momentarily) to conclude that $v'' \preceq^{\mathrm{val}} \langle\!\langle F \rangle\!\rangle[(y : \langle\!\langle\delta\rangle\!\rangle).E^*[y], E'']$, so by (2), $\theta\{x \mapsto v''\} \preceq^{\mathrm{val}}_\Gamma \Theta\{\langle\!\langle F \rangle\!\rangle[(y : \langle\!\langle\delta\rangle\!\rangle).E^*[y], E'']/x\}$ and so by (1), $e\theta\{x \mapsto v''\} \preceq E\{\Theta\{\langle\!\langle F \rangle\!\rangle[(y : \langle\!\langle\delta\rangle\!\rangle).E^*[y], E'']/x\}\}$. This tells us that

$$\begin{aligned} 1 + n &\leq 1 + (E\{\Theta\{\langle\!\langle F \rangle\!\rangle[(y : \langle\!\langle\delta\rangle\!\rangle).E^*[y], E'']/x\}\})_c \\ &= (1 +_c E\{\Theta\{\langle\!\langle F \rangle\!\rangle[(y : \langle\!\langle\delta\rangle\!\rangle).E^*[y], E'']/x\}\})_c \\ &\leq (E^*[E'])_c \end{aligned}$$

and

$$\begin{aligned} v &\preceq^{\mathrm{val}} (E\{\Theta\{\langle\!\langle F \rangle\!\rangle[(y : \langle\!\langle\delta\rangle\!\rangle).E^*[y], E'']/x\}\})_p \\ &\leq (E^*[E'])_p \end{aligned}$$

as needed.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

We just need to verify the assumptions of Lemma 15:

1. $y : \delta \vdash (\texttt{delay}\,(\texttt{fold}_\delta\, y\,\texttt{of}\, x.e))\theta : \sigma\ \texttt{susp}$ and $y : \langle\!\langle \delta \rangle\!\rangle \vdash E^*[y] : \langle\!\langle \sigma\ \texttt{susp} \rangle\!\rangle$.
2. $\_ \vdash v' : F[\delta]$ and $v' \preceq^{\text{val}} E''$ with derivation $\mathscr{E}$.
3. If $\_ \vdash w_0 : \delta$ and $\mathscr{E}_0 :: w_0 \preceq^{\text{val}} E_0$ is a subderivation of $\mathscr{E}$, then $(\texttt{delay}\,(\texttt{fold}_\delta\, y\,\texttt{of}\, x.e))\theta\{y \mapsto w_0\} \preceq^{\text{val}} E^*\{E_0/y\}$.
4. $\texttt{mapv}_F\, y.(\texttt{delay}\,(\texttt{fold}_\delta\, y\,\texttt{of}\, x.e))\theta\ \texttt{into}\ v' \downarrow v''$

(1), (2), and (4) are immediate. Under the assumptions of (3), we must show that $(\texttt{fold}_\delta\, y\,\texttt{of}\, x.e)\theta\{y \mapsto w_0\} \preceq E^*\{E_0/y\}$. Since $\mathscr{E}_0$ is a subderivation of $\mathscr{E}$, the main induction hypothesis applies. ■

Theorem (Syntactic bounding theorem, Thm. 5)**.** *If* $\Gamma \vdash e : \sigma$ *is in the core language, then* $(\Gamma \vdash e : \sigma) \preceq_\sigma (\langle\!\langle \Gamma \rangle\!\rangle \vdash \|e\| : \|\sigma\|)$.

Proof The proof is by induction on $\Gamma \vdash e : \tau$. Most cases proceed by showing that $e \preceq (c, p)$ for some c and p, where $e \downarrow^n v$. By $(\beta_\times)$, $c \le (c, p)_c$ and $p \le (c, p)_p$, so it suffices to show that $n \le c$ and $v \preceq^{\text{val}} p$, and we take advantage of this fact silently.

CASE: $\Gamma, x : \forall\vec{\alpha}.\sigma \vdash x : \sigma\{\vec{\sigma}/\vec{\alpha}\}$. Fix $\theta \preceq^{\text{val}}_{\Gamma, x:\forall\vec{\alpha}.\sigma} \Theta$; we must show that $x\theta \preceq_{\sigma\{\vec{\sigma}/\vec{\alpha}\}} (0, x)\{\Theta\} = (0, \Theta(x))$. The evaluation of $x\theta$ has the form

$$\frac{}{x\theta \downarrow^0 \theta(x)}$$

The cost bound is immediate. For the value bound we must show that $\theta(x) \preceq^{\text{val}}_{\sigma\{\vec{\sigma}/\vec{\alpha}\}} \Theta(x)$. This follows from the definition of $\theta \preceq^{\text{val}}_{\Gamma, x:\forall\vec{\alpha}.\sigma} \Theta$.

CASE: $\Gamma \vdash (\,) : \texttt{unit}$. Fix $\theta \preceq^{\text{val}}_\Gamma \Theta$; we must show that $(\,)\theta \preceq_{\texttt{unit}} (0, (\,))\{\Theta\} = (0, (\,))$. The evaluation of $(\,)\theta$ has the form

$$\frac{}{(\,)\theta \downarrow^0 (\,)}$$

and we have that (cost) $0 \le 0$ and (value) $(\,) \preceq^{\text{val}} (\,)$ by the definition of $\preceq^{\text{val}}_{\texttt{unit}}$.

CASE: $\Gamma \vdash (e_0, e_1) : \sigma_0 \times \sigma_1$. Fix $\theta \preceq^{\text{val}}_\Gamma \Theta$; we must show that $(e_0, e_1)\theta \preceq (c_0 + c_1, (p_0, p_1))\{\Theta\} = ((c_0 + c_1)\{\Theta\}, (p_0, p_1)\{\Theta\})$, where $\|e_i\| = (c_i, p_i)$. The evaluation of $(e_0, e_1)\theta$ has the form

$$\frac{e_0\theta \downarrow^{n_0} v_0 \qquad e_1\theta \downarrow^{n_1} v_1}{(e_0, e_1)\theta \downarrow^{n_0+n_1} (v_0, v_1)}$$

Cost $n_i \le c_i\{\Theta\}$ by the IH so $n_0 + n_1 \le c_0\{\Theta\} + c_1\{\Theta\} = (c_0 + c_1)\{\Theta\}$.
Value $v_i \preceq^{\text{val}} p_i\{\Theta\}$ by the IH so $(v_0, v_1) \preceq^{\text{val}} (p_0\{\Theta\}, p_1\{\Theta\}) = (p_0, p_1)\{\Theta\}$ by the IH.

CASE: $\Gamma \vdash \pi_i\, e : \sigma_i$. Fix $\theta \preceq^{\text{val}}_\Gamma \Theta$; we must show that $(\pi_i\, e)\theta \preceq (c, \pi_i\, p)\{\Theta\} = (c\{\Theta\}, (\pi_i\, p)\{\Theta\})$, where $\|e\| = (c, p)$. The evaluation of $(\pi_i\, e)\theta$ has the form

$$\frac{e\theta \downarrow^n (v_0, v_1)}{(\pi_i\, e)\theta \downarrow^n v_i}$$

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

Cost $n \leq (c)\{\Theta\}$ by the IH.
Value $(v_0, v_1) \preceq^{\text{val}} p\{\Theta\}$ by the IH, so $v_i \preceq^{\text{val}} \pi_i\,(p\{\Theta\}) = (\pi_i\,p)\{\Theta\}$ by the definition of $\preceq^{\text{val}}_{\sigma_0 \times \sigma_1}$.

CASE: $\Gamma \vdash \iota_i\, e : \sigma_0 + \sigma_1$. Fix $\theta \preceq^{\text{val}}_{\Gamma} \Theta$; we must show that $(\iota_i\, e)\theta \preceq (c, \iota_i\, p)\{\Theta\} = (c\{\Theta\}, (\iota_i\, p)\{\Theta\})$, where $\|e\| = (c, p)$. The evaluation of $(\iota_i\, e)\theta$ has the form

$$\frac{e\theta \downarrow^n v}{(\iota_i\, e)\theta \downarrow^n \iota_i\, v}$$

Cost $n \leq c\{\Theta\}$ by the IH.
Value $v \preceq^{\text{val}} p\{\Theta\}$ by the IH, and $\iota_i\,(p\{\Theta\}) \leq \iota_i\,(p\{\Theta\})$, so $\iota_i\, v \preceq^{\text{val}} \iota_i\,(p\{\Theta\}) = (\iota_i\, p)\{\Theta\}$ by the definition of $\preceq^{\text{val}}_{\sigma_0 + \sigma_1}$.

CASE: $\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{x.e_i\}_{i=0,1} : \sigma$. Fix $\theta \preceq^{\text{val}}_{\Gamma} \Theta$; we must show that $(\texttt{case}\ e\ \texttt{of}\ \{x.e_i\}_{i=0,1})\theta \preceq (c +_c \mathsf{case}\, p\ \mathsf{of}\ \{(x : \langle\!\langle \sigma_i \rangle\!\rangle).(c_i, p_i)\}_{i=0,1})\{\Theta\} = c\{\Theta\} +_c \mathsf{case}\, p\{\Theta\}\ \mathsf{of}\ \{(x : \langle\!\langle \sigma_i \rangle\!\rangle).(c_i, p_i)\{\Theta - x\}\}_{i=0,1}$, where $\|e\| = (c, p)$ and $\|e_i\| = (c_i, p_i)$. The evaluation of $(\texttt{case}\ e\ \texttt{of}\ \{x.e_i\}_{i=0,1})\theta$ has the form

$$\frac{e\theta \downarrow^n \iota_i\, v \qquad e_i\theta\{x \mapsto v\} \downarrow^{n_i} v_i}{(\texttt{case}\ e\ \texttt{of}\ \{x.e_i\}_{i=0,1})\theta \downarrow^{n+n_i} v_i}$$

By the IH for e, $\iota_i\, v \preceq^{\text{val}} p\{\Theta\}$, so there is some E' such that $v \preceq^{\text{val}} E'$ and $\iota_i\, E' \leq p\{\Theta\}$. If we set $\theta' = \theta\{v \mapsto x\}$ and $\Theta' = \Theta\{E'/x\}$, then $\theta' \preceq^{\text{val}} \Theta'$, so by the IH for e_i, $e_i\theta' \preceq (c_i\{\Theta'\}, p_i\{\Theta'\})$. Since $\iota_i\, E' \leq p\{\Theta\}$, we have

$$\begin{aligned}(c_i\{\Theta'\}, p_i\{\Theta'\}) &= (c_i, p_i)\{\Theta'\} \\ &\leq \mathsf{case}\ \iota_i\, E'\ \mathsf{of}\ \{(x : \langle\!\langle \sigma_i \rangle\!\rangle).(c_i, p_i)\{\Theta - x\}\}_{i=0,1} \\ &\leq \mathsf{case}\, p\{\Theta\}\ \mathsf{of}\ \{(x : \langle\!\langle \sigma_i \rangle\!\rangle).(c_i, p_i)\{\Theta - x\}\}_{i=0,1}\end{aligned}$$

and so

$$\begin{aligned}(c\{\Theta\} + c_i\{\Theta'\}, p_i\{\Theta'\}) &= c\{\Theta\} +_c (c_i\{\Theta'\}, p_i\{\Theta'\}) \\ &\leq c\{\Theta\} +_c \mathsf{case}\, p\{\Theta\}\ \mathsf{of}\ \{(x : \langle\!\langle \sigma_i \rangle\!\rangle).(c_i, p_i)\{\Theta - x\}\}_{i=0,1}\end{aligned}$$

which we use to complete the next set of calculations.

Cost $n \leq c\{\Theta\}$ and $n_i \leq c_i\{\Theta'\}$, so

$$\begin{aligned}n + n_i &\leq c\{\Theta\} + c_i\{\Theta'\} \\ &\leq (c\{\Theta\} + c_i\{\Theta'\}, p_i\{\Theta'\})_c \\ &\leq (c\{\Theta\} +_c \mathsf{case}\, p\{\Theta\}\ \mathsf{of}\ \{(x : \langle\!\langle \sigma_i \rangle\!\rangle).(c_i, p_i)\{\Theta - x\}\}_{i=0,1})_c.\end{aligned}$$

Value

$$\begin{aligned}v_i &\preceq^{\text{val}} p_i\{\Theta'\} \\ &\leq (c\{\Theta\} + c_i\{\Theta'\}, p_i\{\Theta'\})_p \\ &\leq (c\{\Theta\} +_c \mathsf{case}\, p\{\Theta\}\ \mathsf{of}\ \{(x : \langle\!\langle \sigma_i \rangle\!\rangle).(c_i, p_i)\{\Theta - x\}\}_{i=0,1})_p.\end{aligned}$$

CASE: $\Gamma \vdash \lambda x.e : \sigma' \to \sigma$. Fix $\theta \preceq^{\text{val}}_{\Gamma} \Theta$; we must show that $(\lambda x.e)\theta \preceq (0, \lambda(x : \langle\!\langle \sigma' \rangle\!\rangle).\|e\|)\{\Theta\} = (0, \lambda(x : \langle\!\langle \sigma' \rangle\!\rangle).\|e\|\{\Theta - x\})$. The evaluation of $(\lambda x.e)\theta$ has the form

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

$$\frac{}{(\lambda x.e)\theta \downarrow^0 (\lambda x.e)\theta}$$

so the cost claim is immediate.

Value Fix any $v' \preceq^{\text{val}} E'$. We must show that $e\theta\{x \mapsto v'\} \preceq (\lambda(x : \langle\!\langle \sigma' \rangle\!\rangle).\|e\|\{\Theta - x\})E'$; by definition, $(\beta_{\to})$, and Weakening, it suffices to show that $e\theta\{x \mapsto v'\} \preceq \|e\|\{\Theta\{x \mapsto E'\}\}$. Since $\theta \preceq^{\text{val}} \Theta$ and $v \preceq^{\text{val}} E'$, this follows from the induction hypothesis.

CASE: $\Gamma \vdash e_0\, e_1 : \sigma$. Fix $\theta \preceq^{\text{val}} \Theta$. We must show that $(e_0\, e_1)\theta \preceq ((c_0 + c_1) +_c p_0\, p_1)\{\Theta\}$, where $\|e_i\| = (c_i, p_i)$. The evaluation of $(e_0\, e_1)\theta$ has the form

$$\frac{e_0\theta \downarrow^{n_0} (\lambda x.e_0')\theta' \qquad e_1\theta \downarrow^{n_1} v_1 \qquad e_0'\theta'\{x \mapsto v_1\} \downarrow^{n} v}{(e_0\, e_1)\theta \downarrow^{n_0+n_1+n} v}$$

By the IH, $n_0 \preceq c_0\{\Theta\}$, $(\lambda x.e_0')\theta' \preceq^{\text{val}} p_0\{\Theta\}$, $n_1 \preceq c_1\{\Theta\}$, and $v_1 \preceq^{\text{val}} p_1\{\Theta\}$. By definition of $\preceq^{\text{val}}$, $e_0'\theta'\{x \mapsto v_1\} \preceq (p_0\{\Theta\})(p_1\{\Theta\}) = (p_0\, p_1)\{\Theta\}$, so $n \preceq ((p_0\, p_1)\{\Theta\})_c$ and $v \preceq^{\text{val}} ((p_0\, p_1)\{\Theta\})_p$.

Cost $n_0 + n_1 + n \preceq c_0\{\Theta\} + c_1\{\Theta\} + ((p_0\, p_1)\{\Theta\})_c \preceq (((c_0 + c_1) +_c p_0\, p_1)\{\Theta\})_c$.
Value $v \preceq^{\text{val}} ((p_0\, p_1)\{\Theta\})_p \preceq (((c_0 + c_1) +_c p_0\, p_1)\{\Theta\})_p$.

CASE: $\Gamma \vdash \texttt{delay}\, e : \sigma\ \texttt{susp}$. Fix $\theta \preceq^{\text{val}} \Theta$. We must show that $(\texttt{delay}\, e)\theta \preceq (0, \|e\|)\{\Theta\} = (0, \|e\|\{\Theta\})$. The evaluation of $(\texttt{delay}\, e)\theta$ has the form

$$\frac{}{(\texttt{delay}\, e)\theta \downarrow^0 (\texttt{delay}\, e)\theta}$$

so (cost) $0 \preceq 0$ and (value) since $e\theta \preceq \|e\|\{\Theta\}$ by the IH, $(\texttt{delay}\, e)\theta \preceq^{\text{val}} \|e\|\{\Theta\}$ by the definition of $\preceq^{\text{val}}_{\sigma\ \texttt{susp}}$.

CASE: $\Gamma \vdash \texttt{force}\, e : \sigma$. Fix $\theta \preceq^{\text{val}} \Theta$. We must show that $(\texttt{force}\, e)\theta \preceq (c +_c p)\{\Theta\}$, where $\|e\| = (c, p)$. The evaluation of $(\texttt{force}\, e)\theta$ has the form

$$\frac{e\theta \downarrow^{n} (\texttt{delay}\, e')\theta' \qquad e'\theta' \downarrow^{n'} v}{(\texttt{force}\, e)\theta \downarrow^{n+n'} v}$$

By the IH, $n \preceq c\{\Theta\}$ and $(\texttt{delay}\, e')\theta' \preceq^{\text{val}} p\{\Theta\}$, so by definition of $\preceq^{\text{val}}$, $e'\theta' \preceq p\{\Theta\}$ and hence $n' \preceq (p\{\Theta\})_c$ and $v \preceq^{\text{val}} (p\{\Theta\})_p$. So (cost) $n + n' \preceq c\{\Theta\} + p\{\Theta\}_c \preceq ((c +_c p)\{\Theta\})_c$ and (value) $v \preceq^{\text{val}} p\{\Theta\}_p \preceq ((c +_c p)\{\Theta\})_p$.

CASE: $\Gamma \vdash \mathtt{c}_\delta\, e : \delta$. Fix $\theta \preceq^{\text{val}} \Theta$. We must show that $(\mathtt{c}_\delta\, e)\theta \preceq (c, \mathtt{c}_{\langle\!\langle\delta\rangle\!\rangle}\, p)\{\Theta\}$, where $\|e\| = (c, p)$. The evaluation of $(\mathtt{c}_\delta\, e)\theta$ has the form

$$\frac{e\theta \downarrow^{n} v}{(\mathtt{c}_\delta\, e)\theta \downarrow^{n} \mathtt{c}_\delta\, v}$$

Cost $n \preceq c\{\Theta\}$ by the IH.
Value By the IH we have that $v \preceq^{\text{val}}_{\langle\!\langle F[\delta]\rangle\!\rangle} p\{\Theta\}$, and so by Lemma 2, $v \preceq^{\text{val}}_{F,\delta} p\{\Theta\}$. Since $\mathtt{c}_{\langle\!\langle\delta\rangle\!\rangle}\, p \preceq \mathtt{c}_{\langle\!\langle\delta\rangle\!\rangle}\, p$, the value bound follows by definition of $\preceq^{\text{val}}_\delta$.

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

CASE: $\Gamma \vdash \mathtt{d}_\delta\ e : F[\delta]$. Fix $\theta \preceq^{\mathrm{val}} \Theta$. We must show that $(\mathtt{d}_\delta\ e)\theta \preceq (c, \mathtt{d}_{\langle\!\langle\delta\rangle\!\rangle}\ p)\{\Theta\} = (c\{\Theta\}, \mathtt{d}_{\langle\!\langle\delta\rangle\!\rangle}\ (p\{\Theta\}))$, where $\|e\| = (c, p)$. The evaluation of $(\mathtt{d}_\delta\ e)\theta$ has the form

$$\frac{e\theta \downarrow^n \mathtt{c}_\delta\ v}{(\mathtt{d}_\delta\ e)\theta \downarrow^n v}$$

Cost $n \leq c\{\Theta\}$ by the IH.

Value By the IH, $\mathtt{c}_\delta\ v \preceq^{\mathrm{val}} p\{\Theta\}$, and so by definition of $\preceq^{\mathrm{val}}_\delta$, there is E such that $v \preceq^{\mathrm{val}}_{F,\delta} E$ and $\mathtt{c}_{\langle\!\langle\delta\rangle\!\rangle}\ E \leq p\{\Theta\}$. This latter fact along with the axioms for $\leq$ tell us that $E \leq \mathtt{d}_{\langle\!\langle\delta\rangle\!\rangle}\ (\mathtt{c}_{\langle\!\langle\delta\rangle\!\rangle}\ E) \leq \mathtt{d}_{\langle\!\langle\delta\rangle\!\rangle}\ (p\{\Theta\})$.

CASE: $\Gamma \vdash \mathtt{fold}_\delta\ e'\ \mathtt{of}\ x.e : \sigma$. The type derivation has the form

$$\frac{\Gamma \vdash e' : \delta \qquad \Gamma, x : F[\sigma\ \mathtt{susp}] \vdash e : \sigma}{\Gamma \vdash \mathtt{fold}_\delta\ e'\ \mathtt{of}\ x.e : \sigma}$$

Fix $\theta \preceq^{\mathrm{val}}_\Gamma \Theta$ and without loss of generality assume that $x \notin \mathrm{dom}\,\Gamma \cup \mathrm{dom}\,\theta \cup \mathrm{dom}\,\Theta$; we must show that $(\mathtt{fold}_\delta\ e'\ \mathtt{of}\ x.e)\theta \preceq c' +_c \mathsf{fold}_{\langle\!\langle\delta\rangle\!\rangle}\ p'\ \mathsf{of}\ (x : \langle\!\langle F\rangle\!\rangle[\|\sigma\|]).1 +_c \|e\|$ where $\|e'\| = (c', p')$. The evaluation of $(\mathtt{fold}_\delta\ e'\ \mathtt{of}\ x.e)\theta$ has the form

$$\frac{e'\theta \downarrow^{n'} \mathtt{c}_\delta\ v' \qquad \mathtt{mapv}_F\ y.(\mathtt{delay}\ (\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e))\theta\ \mathtt{into}\ v' \downarrow v'' \qquad e\theta\{x \mapsto v''\} \downarrow^n v}{(\mathtt{fold}_\delta\ e'\ \mathtt{of}\ x.e)\theta \downarrow^{n'+n+1} v}$$

and so the following is also an evaluation, where we write θ' for $\theta\{z \mapsto \mathtt{c}_\delta\ v'\}$:

$$\frac{z\theta' \downarrow^0 \mathtt{c}_\delta\ v' \qquad \mathtt{mapv}_F\ y.(\mathtt{delay}\ (\mathtt{fold}_\delta\ y\ \mathtt{of}\ x.e))\theta'\ \mathtt{into}\ v' \downarrow v'' \qquad e\theta'\{x \mapsto v''\} \downarrow^n v}{(\mathtt{fold}_\delta\ z\ \mathtt{of}\ x.e)\theta' \downarrow^{n+1} v}$$

The IH for e' tells us that $n' \leq c'\{\Theta\}$ and $\mathtt{c}_\delta\ v' \preceq^{\mathrm{val}} p'\{\Theta\}$; combined with the IH for e, Lemma 16 tells us that $(\mathtt{fold}_\delta\ z\ \mathtt{of}\ x.e)\theta' \preceq \mathsf{fold}_{\langle\!\langle\delta\rangle\!\rangle}\ p'\{\Theta\}\ \mathsf{of}\ (x : \langle\!\langle F\rangle\!\rangle[\|\sigma\|]).1 +_c \|e\|\{\Theta\}$ as required.

CASE: $\Gamma \vdash \mathtt{let}\ x = e'\ \mathtt{in}\ e : \sigma$. The type derivation has the form

$$\frac{\Gamma \vdash e' : \sigma' \qquad \Gamma, x : \forall\vec{\alpha}.\sigma' \vdash e : \sigma \qquad \vec{\alpha}\ \text{not free in any}\ \Gamma(y)}{\Gamma \vdash \mathtt{let}\ x = e'\ \mathtt{in}\ e : \sigma}$$

Fix $\theta \preceq^{\mathrm{val}}_\Gamma \Theta$ and without loss of generality assume x is fresh for Γ, θ, and Θ and that no α_i is free in any $\Theta(y)$. We must show that $(\mathtt{let}\ x = e'\ \mathtt{in}\ e)\theta \preceq (c' +_c \|e\|\{\Lambda\vec{\alpha}.p'/x\})\{\Theta\} = c'\{\Theta\} +_c \|e\|\{\Theta\}\{\Lambda\vec{\alpha}.p'\{\Theta\}/x\}$ where $\|e'\| = (c', p')$. The evaluation has the form

$$\frac{e'\theta \downarrow^{n'} v' \qquad e\theta\{x \mapsto v'\} \downarrow^n v}{(\mathtt{let}\ x = e'\ \mathtt{in}\ e)\theta \downarrow^{n'+n} v}$$

The IH for e' tells us that $n' \leq c'\{\Theta\}$ and $v' \preceq^{\mathrm{val}}_{\sigma'} p'\{\Theta\}$. If we can show that $v' \preceq^{\mathrm{val}}_{\forall\vec{\alpha}.\sigma'} \Lambda\vec{\alpha}.p'\{\Theta\}$, then the induction hypothesis applied to e provides the

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press

remaining pieces of the argument. For this we need to show that for any closed $\vec{\rho}$, $v' \preceq^{\mathrm{val}}_{\sigma'\{\vec{\rho}/\vec{\alpha}\}} (\Lambda\vec{\alpha}.p'\{\Theta\})\,\langle\!\langle\vec{\rho}\rangle\!\rangle$, and by $(\beta_\forall)$ and weakening, it suffices to show $v' \preceq^{\mathrm{val}}_{\sigma'\{\vec{\rho}/\vec{\alpha}\}} p'\{\Theta\}\{\langle\!\langle\vec{\rho}\rangle\!\rangle/\vec{\alpha}\}$. This in turn requires us to show that if $\mathrm{ftv}(\sigma) = \{\vec{\alpha}, \vec{\beta}\}$, then for any closed $\vec{\rho'}$, $v' \preceq^{\mathrm{val}}_{\sigma'\{\vec{\rho},\vec{\rho'}/\vec{\alpha},\vec{\beta}\}} p'\{\Theta\}\{\langle\!\langle\vec{\rho},\vec{\rho'}\rangle\!\rangle/\vec{\alpha},\vec{\beta}\}$, which follows from the fact that $v' \preceq^{\mathrm{val}}_{\sigma'} p'\{\Theta\}$.

■

https://doi.org/10.1017/S095679682200003X Published online by Cambridge University Press