A Static Cost Analysis for a Higher-order Language

Norman Danner   Jennifer Paykin [*]

Department of Mathematics and Computer Science
Wesleyan University
Middletown, CT 06459 USA
{ndanner, jpaykin}@wesleyan.edu

James S. Royer

Department of Electrical Eng. and Computer Science
Syracuse University
Syracuse, NY 13244 USA
jsroyer@syr.edu

Abstract

We develop a static complexity analysis for a higher-order functional language with structural list recursion. The complexity of an expression is a pair consisting of a cost and a potential. The former is defined to be the size of the expression's evaluation derivation in a standard big-step operational semantics. The latter is a measure of the "future" cost of using the value of that expression. A translation function $\|\cdot\|$ maps target expressions to complexities. Our main result is the following Soundness Theorem: If t is a term in the target language, then the cost component of $\|t\|$ is an upper bound on the cost of evaluating t. The proof of the Soundness Theorem is formalized in Coq, providing certified upper bounds on the cost of any expression in the target language.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.2.m [*Analysis of Algorithms and Problem Complexity*]: Miscellaneous; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Theory, Verification.

Keywords Higher-order complexity; automated theorem proving; certified bounds.

1. Introduction

Though cost analyses are well-studied, they are traditionally performed by hand in a relatively ad-hoc manner. Formalisms for (partially) automating the analysis of higher-order functional languages have been developed by, e.g., Shultis [1985], Sands [1990], Van Stone [2003], and Benzinger [2004].[1] These formalisms map target-language programs into a domain of complexities, which can then be reasoned about more-or-less formally. The translations carry over information regarding the values of subexpressions, which can then be "discarded" during reasoning about cost should that be appropriate.

In this paper we aim for a similar goal, but take an approach inspired by the work of [Danner and Royer 2009]. There we analyzed a programming language for type-level 2 functions in a restricted type system motivated by work in implicit computational complexity. The goal there was to prove that all programs in our formalism are computable in type-2 polynomial time. Here we take the same analysis tools and apply them to a version of Gödel's System T, not to establish fixed time bounds, but to construct expressions that bound the cost of the given program. The key difference between our approach and those discussed above is that we aim for upper bounds on cost in terms of input size, rather than an exact analysis in terms of values. This more modest goal allows us to develop a notion of complexity that bounds the runtime cost of any target program evaluation on inputs of a given size. Besides providing us with a simpler setting in which to reason about complexity, the absence of values in the upper bounds is also in line with typical algorithm analysis. A long-term goal is to incorporate well-established analysis techniques into the formal system of reasoning that we present here.

In this paper we consider a higher-order language, defined in Section 2, over integers and integer lists with structural list recursion. Since the interesting analyses are done in terms of the sizes of the lists involved, we declare all integers to be of some constant size, which we denote by 1. As an example of our analysis, let us consider list insertion, defined by

```
ins (x, nil) = [x]
ins (x, y::ys) =
    if (x<=y) then x::y::ys else y::(ins(x, ys))
```

A simple cost analysis of `ins` yields a recursive cost function `ins_c` for which the recurrence argument is the length of the list argument of `ins`:

$$\texttt{ins\_c}(0) = c_0 \qquad \texttt{ins\_c}(n+1) = c_1 + \big(c_2 \vee (c_3 + \texttt{ins\_c}(n))\big)$$

where the c_i are constants. The maximum in the recursion clause ensures that we need not consider the value of the test `x <= y`. An easily-formalized proof by induction tells us that $\texttt{ins\_c}(n) \in O(n)$. But this is not enough for our purposes. We want our analyses to be compositional, so that we can directly use the analysis of, say, `ins` in the analysis of insertion-sort defined by

```
ins_sort xs = fold ins xs nil
```

An implicit part of the analysis of `ins_sort` uses the *size* of `ins xs` in terms of the size of `xs`. Thus in addition to the cost analysis above, our approach also generates a size analysis, which we refer to as potential (more on this terminology momentarily). The following is

[*] Current address: Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104 USA.

[1] This is an incomplete list; we focus here only on work that directly addresses the analysis of higher-order functional languages and/or the automation of that analysis, and we discuss these systems in more detail in Section 7.

a possible potential analysis of ins:

$$\texttt{ins\_p}(0) = 1 \qquad \texttt{ins\_p}(n+1) = (n+1) \vee (1 + \texttt{ins\_p}(n))$$

where again we take a maximum so that we need not consider the value of the test `x <= y`.

Since we are interested in higher-order languages, we also need to consider algorithms which take functions as input. For this, we need some way to represent the cost associated with using a function; this, combined with size, is our notion of *potential*, which we describe more fully in Section 3. For type-level 0 objects, we can think of potential as ordinary size, which is why ins_p is a function from integers to integers. The potential of a function essentially encompasses its cost analysis—the potential of f is a map from potentials p (representing the potential of the argument) to the complexity of applying f to an argument of size p. Our complexity analysis of an expression yields a pair consisting of a cost (of computing that expression to obtain a value) and a potential (representing the size of that value). We refer to such a pair as a *complexity*.

The complexity language that we define in Section 3 allows us to express recurrences like ins_c and ins_p. In Section 4 we define a translation function $\|\cdot\|$ from the target language to the complexity language so that $\|\texttt{ins}\|$ encompasses these two recurrences. The recurrences that result from the translation will be more complicated than those shown here because they take complexities, as opposed to sizes, as arguments. But as shown in Section 4.1.1, it is easy to extract ins_c and ins_p from $\|\texttt{ins}\|$.

As another example, consider the map function defined by

```
map h nil = nil
map h (x :: xs) = (h x) :: (map h xs)
```

The natural cost analysis of map yields a recurrence that depends on the cost of applying h to account for the term h x in the recursion clause. Note that this is *not* the cost of h itself (or any specific function argument). Indeed, since any specific function argument is likely to be expressed as a λ-abstraction, the cost of such an argument would be 1. Instead, we must refer to the *potential* of the function argument h; applied to the potential of a list element x (representing the size of x), we obtain the complexity of the application $h(x)$. That complexity is a pair consisting of the cost of the application and its potential. Taking into account that our integers have constant size 1, we end up with a cost analysis that looks something like the following, where h now represents the *complexity* of the function argument:[2]

$$\texttt{map\_c}(h,0) = 0 \qquad \texttt{map\_c}(h,n+1) = (h_p(1))_c + \texttt{map\_c}(h,n)$$

where $(\cdot)_c$ and $(\cdot)_p$ extract the cost and potential components of a complexity, respectively. The potential analysis is straightforward and does not depend on the function argument:

$$\texttt{map\_p}(h,0) = 0 \qquad \texttt{map\_p}(h,n+1) = 1 + \texttt{map\_p}(h,n)$$

It is nice that our translation gives the expected costs in these examples, but of course we want to know that it is sound, in the sense that $\|t\|$ bounds the complexity of t. We prove such a Soundness Theorem in Section 5 and state clearly the corollary that an upper bound on the evaluation of cost t can be derived directly from $\|t\|$.

The function $\|\cdot\|$ is computable, and hence provides a formal link between the program source code and its complexity bound. We have implemented a subset of the target and complexity languages, translation, and proof of the Soundness Theorem in Coq, and we discuss some of the details of this in Section 6. The formalization thus provides a mechanism for providing *certified* upper bounds on the complexity of target language expressions. This is in distinction to a more traditional ad-hoc analysis; such an analysis, even if formalized in a system such as Coq, is not tied to the source code in a machine-checkable manner, and so one still does not have a proof that the code to be executed satisfies the cost bounds that are asserted.

[2] As we will see, parameters in complexity functions that correspond to (list-) recursion arguments represent the potential of such arguments. Parameters that correspond to non-recursion arguments represent the complexity of those arguments.

2. Target language

The target language is essentially a variant on Gödel's System T; its syntax, typing, and operational semantics are given in Figures 1–3. The language provides for higher-order programming over integers, booleans, and integer lists (the *base types*). The latter are defined as a recursive datatype, and the datatype definition automatically provides for structural recursion. The work in this paper extends to other comparable recursive datatypes; we treat the special case here to cut down on notation.[3] Since the language is straightforward, we omit many of the details. A *term* is a typeable expression $\Gamma \vdash t : \tau$. The operational semantics defines a big-step call-by-value evaluation relation that relates *closures* $(\Gamma \vdash t : \tau)\xi$ to *values* $v\,\theta$ (we usually drop the typing details and just write $t\,\xi$). A closure $t\,\xi$ consists of a term t and a *value environment* ξ that maps variables to values such that Dom ξ contains all of t's free variables. We write $\{\}$ for the empty environment. A value $v\,\theta$ consists of a *value expression* and a value environment. A value expression is any of the following:

- A boolean value *tt* or *ff* or integer value $n \in \mathbf{Z}$;
- Any finite sequence $(n_0, \dots, n_{k-1})$ of integers;
- Any expression of the form $\lambda x.r$.

We often write (n, ns) for the sequence $(n, n_0, \dots, n_{k-1})$ when $ns = (n_0, \dots, n_{k-1})$ and write $()$ for the empty sequence. Since the grammar does not allow lists of higher-order expressions and the semantics does not have side-effects, we can safely drop the environment at leaves of evaluation derivations that derive values of base type. We do so in order to simplify the statements of the rules for evaluating lists; it is not necessary in practice. To further clean up notation, we often write *tt* or *ns* instead of $tt\{\}$ or $ns\{\}$.

The evaluation of $\texttt{fold}\,r\,\texttt{of}\,(s, [x, xs, w]t)$ when $r\xi \downarrow (n, ns)$ deserves some comment. A more natural rule might be

$$\frac{r\xi \downarrow (n,ns) \quad (\texttt{fold}\,ns\,\texttt{of}\,(s,[x,xs,w]t))\xi \downarrow v_0\,\theta_0 \quad t\,\xi_1 \downarrow v\,\theta}{(\texttt{fold}\,r\,\texttt{of}\,(s,[x,xs,w]t))\xi \downarrow v\,\theta}$$

However, the fold "expression" in the hypothesis is not well-formed, because *ns* is not an expression, it is a value. There is an obvious isomorphism between the two that we could employ, but then evaluating the hypothesis fold expression would require re-evaluating the list corresponding to *ns* which would add (possibly non-trivially) to the cost of the evaluation. Thus we choose a fresh variable y and bind y to *ns* in the environment; as a result, every future evaluation of the recursion argument will have cost 1.[4]

We define the *cost* of a closure $t\xi$, cost$(t\xi)$, to be the size of the evaluation derivation of $t\xi$ (it is straightforward to prove that derivations are unique). We charge unit cost for arithmetic operations and count every inference rule. We can easily adapt the system for other notions of cost (e.g., counting only creation of cons-cells) by modifying the complexity semantics described in Section 3.

[3] Our language does not support general recursion, and recursive datatypes in general pose some difficulties; we discuss both issues in Section 8.

[4] If fold were defined in terms of a more general letrec constructor, then a standard operational semantics of letrec as in [Reynolds 1998] would introduce a comparable fresh variable.

$$\frac{}{x\xi \downarrow \xi(x)} \qquad \frac{}{c\xi \downarrow \underline{c}\{\}} \qquad \frac{}{(\lambda x.r)\xi \downarrow (\lambda x.r)\xi}$$

$$\frac{r\xi \downarrow n_r\{\} \qquad s\xi \downarrow n_s\{\} \qquad n_r\, R\, n_s}{(r\, R\, s)\xi \downarrow tt\{\}}\ (R = <, \leq, =, \ldots) \qquad \frac{r\xi \downarrow n_r\{\} \qquad s\xi \downarrow n_s\{\} \qquad \neg(n_r\, R\, n_s)}{(r\, R\, s)\xi \downarrow ff\{\}}\ (R = <, \leq, =, \ldots)$$

$$\frac{r\xi \downarrow n_r\{\} \qquad s\xi \downarrow n_s\{\} \qquad n = n_r \bullet n_s}{(r \bullet s)\xi \downarrow n\{\}}\ (\bullet = +, -, \times, \ldots)$$

$$\frac{}{\texttt{nil}\,\xi \downarrow ()\{\}} \qquad \frac{r\xi \downarrow n\{\} \qquad s\xi \downarrow ns\{\}}{(r :: s)\xi \downarrow (n, ns)\{\}}$$

$$\frac{r\xi \downarrow (\lambda x.r_0)\theta_0 \qquad s\xi \downarrow v_1\,\theta_1 \qquad r_0\,\theta_0\{x \mapsto v_1\,\theta_1\} \downarrow v\,\theta}{(r s)\xi \downarrow v\,\theta}$$

$$\frac{r\xi \downarrow ()\{\} \qquad s\xi \downarrow v\,\theta}{(\texttt{case}\ r\ \texttt{of}\ (s, [x, xs]t))\xi \downarrow v\,\theta} \qquad \frac{r\xi \downarrow (n, ns)\{\} \qquad t\,\xi\{x, xs \mapsto n, ns\} \downarrow v\,\theta}{(\texttt{case}\ r\ \texttt{of}\ (s, [x, xs]t))\xi \downarrow v\,\theta}$$

$$\frac{r\xi \downarrow ()\{\} \qquad s\xi \downarrow v\,\theta}{(\texttt{fold}\ r\ \texttt{of}\ (s, [x, xs, w]t))\xi \downarrow v\,\theta} \qquad \frac{r\xi \downarrow (n, ns)\{\} \quad (\texttt{fold}\ y\ \texttt{of}\ (s, [x, xs, w]t))\xi_0 \downarrow v_0\,\theta_0 \quad t\,\xi_1 \downarrow v\,\theta}{(\texttt{fold}\ r\ \texttt{of}\ (s, [x, xs, w]t))\xi \downarrow v\,\theta}$$

$$\text{where } y \text{ is fresh};\ \xi_0 = \xi\{y \mapsto ns\};\ \xi_1 = \xi\{x, xs, w \mapsto n, ns, v_0\,\theta_0\}$$

Figure 3. Target language operational semantics. c ranges over integer and boolean constants and $\underline{c}$ over the corresponding values. The third hypothesis of the relation and operation rules has unit cost.

$$\begin{aligned} e \in \text{Exp} ::= {} & X \mid Z \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid e :: e \mid e\, R\, e \mid \\ & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \lambda x.e \mid e\, e \\ & \texttt{case}\ e\ \texttt{of}\ (e, [x, xs]e) \mid \texttt{fold}\ e\ \texttt{of}\ (e, [x, xs, w]e) \end{aligned}$$

Figure 1. Target language grammar. X ranges over variable identifiers, Z over integer constants, and R over numerical binary relations such as $\leq$ and $>$ and numerical binary operators such as $+$ and $\times$.

$$\sigma, \tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{int*} \mid \sigma \to \tau$$
$$\texttt{int*} ::= \texttt{nil} \mid ::\ \texttt{of}\ (\texttt{int}, \texttt{int*})$$

$$\frac{\Gamma \vdash e_0 : \texttt{int*} \qquad \Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \texttt{int}, xs : \texttt{int*} \vdash e_2 : \sigma}{\Gamma \vdash \texttt{case}\ e_0\ \texttt{of}\ (e_1, [x, xs]e_2) : \sigma}$$
$$\frac{\Gamma \vdash e_0 : \texttt{int*} \quad \Gamma \vdash e_1 : \sigma \quad \Gamma, x : \texttt{int}, xs : \texttt{int*}, w : \sigma \vdash e_2 : \sigma}{\Gamma \vdash \texttt{fold}\ e_0\ \texttt{of}\ (e_1, [x, xs, w]e_2) : \sigma}$$

Figure 2. Target language types and typing. Rules not shown here are the expected ones.

3. The complexity language

Our goal is to assign a complexity $\|t\|$ to each target language expression t. We have three desiderata on $\|\cdot\|$:

- $\|t\|$ must provide an upper bound on the cost of evaluating t.
- $\|\cdot\|$ must be compositional.
- $\|t\|$ must not depend on the value of any subexpression of t.

Since closures, not expressions, are evaluated in the target language, $\|t\|$ will also be an expression, the meaning of which is determined by an environment assigning complexities to its free variables. To say that $\|\cdot\|$ is compositional means that $\|t\|$ depends only on expressions $\|s\|$ for subexpressions s of t. Because of the third constraint, the information we provide about the evaluation cost of t is not as precise as that of as the systems mentioned in the introduction. However, this constraint is in line with almost all the work in practical analysis of algorithms and thus opens up the possibility of using the considerable body of tools and tricks of analysis of algorithms in our verifications.

A complexity measure that considers only cost is insufficient if we want to be able to handle higher-order expressions like `map`. To see why, consider any expression $\lambda x.r$ such that $\texttt{map}(\lambda x.r)$ is well-typed.[5] Assuming that $\|\cdot\|$ is compositional, if $\|t\|$ were to provide information on just the cost of evaluating t, then since the cost of $\lambda x.r$ is 1, the cost of $\texttt{map}(\lambda x.r)$ would be independent of r. What we need instead is a complexity measure such that $\|t\|$ not only captures the cost of evaluating t, but also the cost of *using* t. We call this latter notion *potential*, and a complexity will be a pair consisting of a cost and a potential. To gain some intuition for the full definition, we consider the type-level 0 and 1 cases. At type-level 0, the potential cost of an expression is a measure of the size of that expression's value. Now consider a type-level 1 expression r. The *use* of r is its application to a type-level 0 expression s. The cost of such an application is the sum of (i) the cost of evaluating r to a value $\lambda x.r'$; (ii) the cost of evaluating s to a value v'; (iii) the cost of evaluating $r'[x \mapsto v']$; and (iv) a "charge" for the inference. Since (iii) depends (in part) on the size of v' (i.e., the potential of s), by compositionality complexities must capture both cost and potential. Furthermore, (iii) is defined in terms of the potential of r (i.e., the potential of $\lambda x.r'$). Thus the potential of a type-level 1 expression should be a map from type-level 0 potentials to type-level 0 complexities, and in general the potential of an expression of type $\sigma \to \tau$ should be a map from potentials of type-σ expressions to complexities of type-τ expressions.

We now turn to the formal definitions given in Figures 4–7. We define a *complexity language* over a simple type structure with products into which our translation function $\|\cdot\|$ will map. The *full complexity types* consist of the simple types with products over $\mathbf{N}$, which is intended to be the natural numbers and represents both costs and potentials of base-type values. The *complexity* and *potential* types are defined by the following mutual induction, following our preceding discussion about the potential of higher-order values:

[5] We phrase this informal discussion in terms of evaluating closed expressions rather than closures.

1. If γ is a potential type, then $\mathbf{N} \times \gamma$ is a complexity type.
2. $\mathbf{N}$ is a potential type.
3. If γ is a potential type and τ a complexity type, then $\gamma \to \tau$ is a potential type.

We introduce two notations for potential types γ: $\gamma^{\Box} = \mathbf{N} \times \gamma$ and $\gamma \to^{\Box} \tau = (\gamma \to \tau)^{\Box}$ (remember that the potential of a function is a map from argument potentials to result complexities).

The complexity expression constructors roughly mirror the target expression constructors, and the meaning of the former is intended to capture the complexity (cost and potential) of the latter. The grammar of complexity expressions is as expected for the product-related types, except we write t_c and t_p for the first and second projections (for "cost" and "potential"), respectively. The usual abstraction and application are replaced by $\lambda_* x.r$ and $r * s$. As we will see, we need an operation that "applies" $\|r\|$ to $\|s\|$; since both are complexities, and hence pairs, ordinary application does not suffice. Thus we introduce the $*$ operator so that we can define $\|rs\| = \|r\| * \|s\|$. $\lambda_* x.r$ is the corresponding abstraction operator. Conditional expressions are eliminated in the complexity language. A conditional expression is translated to (essentially) the maximum of the complexity of its two branches. This matches our interest in upper bounds on complexity, ensuring that the complexity of a conditional bounds the cost of any possible evaluation of that conditional on inputs of a given size or smaller. The `fold` constructor for recursive datatypes has a counterpart `pfold` in the complexity language so that recursive definitions in the target language are translated to recurrences in the complexity language. Instead of branching on a constructor, `pfold` branches potentials. Since `case` can be seen as a trivial version of `fold`, it has a corresponding counterpart `pcase`.[6]

Meanings are assigned to complexity terms (typeable complexity expressions) through a denotational semantics . We take the standard denotation $[\![\cdot]\!]$ of full complexity types (interpreting $\mathbf{N}$ as the natural numbers). If Γ is a full complexity type environment, we say that ξ is Γ-*consistent* if $\xi(x) \in [\![\Gamma(x)]\!]$ for all $x \in \operatorname{Dom}\Gamma$. Finally, we define a function $[\![\cdot]\!]-$ that maps a complexity term $\Gamma \vdash e : \tau$ and Γ-consistent environment ξ to $[\![\Gamma \vdash e : \tau]\!]\xi \in [\![\tau]\!]$. We write $[\![e]\!]\xi$ when Γ is clear from context.

The denotational semantics of the complexity expression constructors describes the cost and potential of the corresponding target-language constructors. For example, consider the evaluation of $t = rs$ (again, we phrase this discussion in terms of closed expressions for clarity):

$$\frac{r \downarrow \lambda x.r' \qquad s \downarrow v' \qquad r'\{x \mapsto v'\} \downarrow v}{rs \downarrow v}.$$

$\operatorname{cost}(r)$ and $\operatorname{cost}(s)$ both contribute to $\operatorname{cost}(t)$. Recalling our earlier discussion of higher-type potentials, if $\|r\|$ and $\|s\|$ are the complexities of r and s, then $\|r\|_p(\|s\|_p)$ (a complexity) gives both the cost of evaluating $r'\{x \mapsto v'\}$ as well as its potential; but its potential is also the potential of t. Thus we define the meaning of $*$ expressions so that

$$\begin{aligned}[\![\|r\| * \|s\|]\!] = \big(1 + [\![\|r\|_c]\!] + [\![\|s\|_c]\!] + \big([\![\|r\|_p]\!]([\![\|s\|_p]\!])\big)_c,\\ \big([\![\|r\|_p]\!]([\![\|s\|_p]\!])\big)_p\big).\end{aligned}$$

We frequently need to "add cost to a complexity," so we define $\operatorname{dally}(n,(c,p)) = (n+c,p)$. Now we can write, for example,

$$[\![\|r\| * \|s\|]\!] = \operatorname{dally}\big(1 + [\![\|r\|_c]\!] + [\![\|s\|_c]\!], [\![\|r\|_p]\!]([\![\|s\|_p]\!])\big).$$

[6] In fact, we could also define conditionals in the target language to be syntactic sugar for a `case` over the trivial recursive datatype of booleans, and we would end up with the same complexity semantics.

$$\begin{aligned} e ::= & X \mid N \mid e+e \mid e \vee e \mid (e,e) \mid e_c \mid e_p \mid \lambda_* x.e \mid e * e \\ & \texttt{pcase}\ e\ \texttt{of}\ (e, [p, ps]e) \mid \texttt{pfold}\ e\ \texttt{of}\ (e, [p, ps, w]e). \end{aligned}$$

Figure 4. Complexity language expressions. X is a set of variables and N a set of constants for each $n \in \mathbf{N}$.

$$\begin{aligned} \sigma, \tau &::= \mathbf{N} \times \gamma \\ \gamma &::= \mathbf{N} \mid \gamma \to \tau \end{aligned}$$

Figure 5. Complexity and potential types. The *full* complexity types consist of the simple types with products over $\mathbf{N}$.

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{}{\Gamma \vdash n : \mathbf{N}}$$

$$\frac{\Gamma \vdash r : \mathbf{N} \qquad \Gamma \vdash s : \mathbf{N}}{\Gamma \vdash r + s : \mathbf{N}} \qquad \frac{\Gamma \vdash r : \tau \qquad \Gamma \vdash s : \tau}{\Gamma \vdash r \vee s : \tau}$$

$$\frac{\Gamma \vdash r : \mathbf{N} \qquad s : \gamma}{\Gamma \vdash (r,s) : \gamma^{\Box}} \qquad \frac{\Gamma \vdash r : \gamma^{\Box}}{\Gamma \vdash r_c : \mathbf{N}} \qquad \frac{\Gamma \vdash r : \gamma^{\Box}}{\Gamma \vdash r_p : \gamma}$$

$$\frac{\Gamma, x : \gamma^{\Box} \vdash r : \eta^{\Box}}{\Gamma \vdash \lambda_* x.r : \gamma \to^{\Box} \eta^{\Box}} \qquad \frac{\Gamma \vdash r : \gamma \to^{\Box} \eta^{\Box} \qquad \Gamma \vdash s : \gamma^{\Box}}{\Gamma \vdash r * s : \eta^{\Box}}$$

$$\frac{\Gamma \vdash r : \mathbf{N} \qquad \Gamma \vdash s : \gamma^{\Box} \qquad \Gamma, p : \mathbf{N}, ps : \mathbf{N} \vdash t : \gamma^{\Box}}{\Gamma \vdash \texttt{pcase}\ r\ \texttt{of}\ (s, [p, ps]t) : \gamma^{\Box}}$$

$$\frac{\Gamma \vdash r : \mathbf{N} \qquad \Gamma \vdash s : \gamma^{\Box} \qquad \Gamma, p : \mathbf{N}, ps : \mathbf{N}, w : \gamma^{\Box} \vdash t : \gamma^{\Box}}{\Gamma \vdash \texttt{pfold}\ r\ \texttt{of}\ (s, [p, ps, w]t) : \gamma^{\Box}}$$

Figure 6. Complexity language typing rules. Recall that $\gamma^{\Box} = \mathbf{N} \times \gamma$ and $\gamma \to^{\Box} \tau = (\gamma \to \tau)^{\Box}$.

`pcase` and `pfold` expressions have a slightly more complex semantics that might be expected. Focusing on the former, the non-zero branch of a `pcase` expression must take the maximum of the two branches, rather than just the second branch, even though a non-zero potential ought to correspond to a non-empty list. Because our goal is to establish upper bounds on complexity (hence potential), it may be that the branching expression in the target expression evaluates to `nil`, but its translation has a non-zero potential. As an example, consider the term t defined by

```
case (if true then nil else [0]) of ([0,0], [x,xs]nil)
```

The test expression translates to a complexity with potential 1; if $[\![\|t\|]\!]$ were to take into account only the non-nil branch, we would conclude that t has complexity $(7,0)$ (cost 7, potential 0), whereas in fact it has cost 9 and size 2. A similar issue arises with `pfold`, although in this case the cost is not an issue; because `pfold` is a structural recursion, the base case expression will be evaluated, and hence its cost included in the total cost (see Lemma 7 for a precise statement).

4. Translation from target to complexity language

The translation from target language to complexity language is given in Figure 8. We assume a bijection between target and complexity variables, so that when we write $\|x\| = x$, the occurrence of x on the left-hand side is a target variable and the occurrence of x on the right-hand side is the corresponding complexity variable. The translation of the simplest target expression constructors gives expressions that describe directly the cost and size of the corresponding target expressions; more complex constructors are translated to the corresponding

$$
\begin{aligned}
[\![\Gamma, x:\sigma \vdash x:\sigma]\!]\xi &= \xi(x)\\
[\![\Gamma \vdash n:\mathbf{N}]\!]\xi &= n\\
[\![\Gamma \vdash r+s:\mathbf{N}]\!]\xi &= [\![r]\!]\xi + [\![s]\!]\xi\\
[\![\Gamma \vdash r\vee s:\tau]\!]\xi &= [\![r]\!]\xi \vee [\![s]\!]\xi\\
[\![\Gamma \vdash (r,s):\gamma^{\Box}]\!]\xi &= ([\![r]\!]\xi, [\![s]\!]\xi)\\
[\![\Gamma \vdash r_c:\mathbf{N}]\!]\xi &= \pi_0([\![r]\!]\xi)\\
[\![\Gamma \vdash r_p:\gamma]\!]\xi &= \pi_1([\![r]\!]\xi)\\
[\![\Gamma \vdash \lambda_* x.r:\gamma \to^{\Box} \eta^{\Box}]\!]\xi &= (1, \lambda\!\!\lambda p.[\![\Gamma, x:\gamma^{\Box} \vdash r:\eta^{\Box}]\!]\xi\{x \mapsto (1,p)\})\\
[\![\Gamma \vdash r * s:\eta^{\Box}]\!]\xi &= \mathrm{dally}(1 + [\![r_c]\!]\xi + [\![s_c]\!]\xi, [\![r_p]\!]\xi([\![s_p]\!]\xi))\\
[\![\Gamma \vdash \texttt{pcase}\ r\ \texttt{of}\ (s,[p,ps]t):\gamma^{\Box}]\!]\xi &= \begin{cases} [\![s]\!]\xi, & [\![r]\!]\xi = 0\\ [\![s]\!]\xi \vee [\![t]\!]\xi_1, & [\![r]\!]\xi = q+1 \end{cases}\\
&\quad \text{where } \xi_1 = \xi\{p, ps \mapsto 1, q\}\\
[\![\Gamma \vdash \texttt{pfold}\ r\ \texttt{of}\ (s,[p,ps,w]t):\gamma^{\Box}]\!]\xi &= \begin{cases} [\![s]\!]\xi, & [\![r]\!]\xi = 0\\ (2 + [\![rec_c]\!]\xi_0 + [\![t_c]\!]\xi_1, & \\ \quad [\![s_p]\!]\xi \vee [\![t_p]\!]\xi_1), & [\![r]\!]\xi = q+1 \end{cases}\\
&\quad \text{where } rec = \texttt{pfold}\ y\ \texttt{of}\ (s,[p,ps,w]t),\ \xi_0 = \xi\{y \mapsto q\},\ \xi_1 = \xi\{p,ps,w \mapsto 1, q, (1, rec_p)\}
\end{aligned}
$$

Figure 7. Denotational semantics of complexity terms. $\lambda\!\!\lambda p.\cdots$ denotes the semantic function that maps p to $\cdots$.

complexity constructors, and we leave it to the denotational semantics to extract the complexities from there. There is a choice to be made regarding which constructors in the target language have corresponding constructors in the complexity language. Certainly `fold` must be in this list, so that recursive programs are mapped to recurrences. But whether abstraction and application should have counterparts, or whether they should be translated into complexity expressions that directly describe the denotational semantics of λ_* and $*$, is not completely clear. The choice we have made seems to allow simpler reasoning about the translated expressions, reasoning that we should be able to easily formalize in Coq, and which would also be more familiar to programmers.

Before proceeding to examples, we note that translation preserves type derivations. For a target type context Γ, define $\|\Gamma\| = \{(x:\|\tau\|) \mid (x:\tau) \in \Gamma\}$.

PROPOSITION 1. *If* $\Gamma \vdash r:\tau$, *then* $\|\Gamma\| \vdash \|r\|:\|\tau\|$.

4.1 Examples

We show the results of translating the insertion-sort and map functions in this section. We freely transform expressions in the complexity language according to validities in the semantics, for example adding natural numbers and computing maximums when possible. The soundness of each such equality or inequality is easily proved, and so such transformations could easily be incorporated in a formalized proof that simplifies $\|t\|$ into a more amenable form.

4.1.1 Insertion-sort

We start by considering the list-insertion function defined by

```
ins = λx.λxs.fold xs of (x :: nil,
          [y, ys, w] if x <= y then x :: y :: ys
                     else y :: w)
```

and showing that it has a linear running time in the size of its input. Translating directly yields

$$
\begin{aligned}
&\|\texttt{ins}\| = \lambda_* x, xs.\texttt{pfold}\, xs_p\ \texttt{of}\ ((2+x_c, 1),\\
&\quad [p,ps,w]\,\mathrm{dally}(4+x_c, (4+x_c, 2+ps) \vee (2+w_c, 1+w_p))).
\end{aligned}
$$

If we write $f_{ins}(x,xs)$ for $\|\texttt{ins}\| * x * xs$, apply the equations for the denotations of λ_* and $*$, and use properties of the order on natural numbers, we have

$$
\begin{aligned}
f_{ins}(x,xs) &= \mathrm{dally}(4 + x_c + xs_c, \texttt{pfold}\, xs_p\ \texttt{of}\ ((3,1),\\
&\qquad [p,ps,w]\,\mathrm{dally}(5,\\
&\qquad\qquad (5, 2+ps) \vee (2+w_c, 1+w_p))))\\
&\le \mathrm{dally}(4 + x_c + xs_c, \texttt{pfold}\, xs_p\ \texttt{of}\ ((3,1),\\
&\qquad [p,ps,w](10+w_c, (2+ps) \vee (1+w_p)))
\end{aligned}
$$

To turn this into a recognizable form, set

$$
\begin{aligned}
g(z) = \texttt{pfold}\ z\ \texttt{of}\ \big((3,1)&\\
[p,ps,w](10+w_c, (2+ps) \vee (1+w_p))\big)&
\end{aligned}
$$

so that $f_{ins}(x,xs) = \mathrm{dally}(4 + x_c + xs_c, g(xs_p))$. Rewriting g as a pair of recurrences (one each for cost and potential) we obtain

$$
\begin{aligned}
g_c(0) &= 3 & g_c(q+1) &= 13 + g_c(q)\\
g_p(0) &= 1 & g_p(q+1) &= (2+q) \vee (1 + g_p(q))
\end{aligned}
$$

Here again we have used the equations from the denotational semantics of `pfold`. For example, the expression for $g_c(q+1)$ is $2 + rec_c + t_c$ where rec is the recursive call (i.e., $g(q)$) and

$$
\begin{aligned}
t = (10 + w_c, (2+ps) \vee (1+w_p))[w \mapsto (1, rec_p)] &=\\
(11, (2+ps) \vee (1+rec_p))&.
\end{aligned}
$$

A straightforward induction establishes $g(z) \le (13z+3, z+1)$ and hence $f_{ins}(x,xs) \le (13xs_p + 7 + x_c + xs_c, xs_p + 1)$.

Continuing, we now consider the insertion-sort function defined by

```
ins_sort = λxs.fold xs of (nil,
               [y, ys, w](insert y w))
```

$$\langle\langle \mathsf{b} \rangle\rangle = \mathbf{N} \qquad \langle\langle \sigma \to \tau \rangle\rangle = \langle\langle \sigma \rangle\rangle \to \|\tau\| \qquad \|\tau\| = \mathbf{N} \times \langle\langle \tau \rangle\rangle = \langle\langle \tau \rangle\rangle^{\Box}$$

$$\begin{aligned}
\|x\| &= x \\
\|c\| &= (1,1) \\
\|r\,R\,s\| &= (2 + \|r\|_c + \|s\|_c, 1) \\
\|\mathtt{nil}\| &= (1,0) \\
\|r :: s\| &= (1 + \|r\|_c + \|s\|_c, 1 + \|s\|_p) \\
\|\mathtt{if}\ r\ \mathtt{then}\ s\ \mathtt{else}\ t\| &= \mathrm{dally}(1 + \|r\|_c, \|s\| \vee \|t\|) \\
\|\lambda x.r\| &= \lambda_* x.\|r\| \\
\|rs\| &= \|r\| * \|s\| \\
\|\mathtt{case}\ r\ \mathtt{of}\ (s, [x, xs]t)\| &= \mathrm{dally}(1 + \|r\|_c, \mathtt{pcase}\ \|r\|_p\ \mathtt{of}\ (\|s\|, [p, ps]\|t\|\{x, xs \mapsto (1,p), (1,ps)\})) \\
\|\mathtt{fold}\ r\ \mathtt{of}\ (s, [x, xs, w]t)\| &= \mathrm{dally}(1 + \|r\|_c, \mathtt{pfold}\ \|t\|_p\ \mathtt{of}\ (\|s\|, [p, ps, w]\|t\|\{x, xs \mapsto (1,p), (1,ps)\}))
\end{aligned}$$

Figure 8. Translation from target types and expressions to complexity types and expressions. c ranges over integer and boolean constants.

Following the approach above, writing $f_{isrt}(xs)$ for $\|\mathtt{ins\_sort}\| * xs$ and $g(z)$ for $\mathtt{pfold}\ z\ \mathtt{of}\ ((1,0), [p, ps, w] f_{isrt}((1,p), w))$, we have

$$f_{isrt}(xs) = \mathrm{dally}(2 + xs_c, g(xs_p))$$
$$g_c(0) = 1 \qquad g_c(q+1) = 2 + g_c(q) + \big(f_{ins}\big((1,1),(1,g_p(q))\big)\big)_c$$
$$g_p(0) = 0 \qquad g_p(q+1) = \big(f_{ins}\big((1,1),(1,g_p(q))\big)\big)_p$$

Using our bound on f_{ins}, a proof by induction to show that $g_p(q) \le q$, and then this last inequality to simplify the bound on $g_c(q+1)$ we have

$$g_c(0) \le 1 \qquad g_c(q+1) \le 11 + g_c(q) + 13q$$
$$g_p(0) \le 0 \qquad g_p(q+1) \le 1 + g_p(q)$$

from which we conclude $g(z) \le (13z^2 + 11z + 1, z)$ and hence $f_{isrt}(xs) \le (13xs_p^2 + 9xs_p + 3 + xs_c, xs_p)$.

Although we don't often think of the analysis of insertion-sort involving an analysis of size (something we are forced to do when working with complexities), in fact such size analyses are usually implicit. For example, in the standard analysis of insertion-sort, we usually implicitly make use of the correctness of the algorithm to assert that the length of $\mathtt{ins\_sort}(xs)$ is the same as the length of xs.

It is also worth noting that the analysis of $\|\mathtt{ins\_sort}\|$ only depends on properties of $\|\mathtt{insert}\|$, namely that $f_{ins}(x, xs) \le (a \cdot xs_p + x_c + xs_c + b, xs_p + 1)$. One way to see this is to note that we could have defined a general fold function

```
list_fold = λf, xs, a.fold xs of (a,
              [y, ys, w](f y w))
```

and analyzed $\|\mathtt{list\_fold}\|$. The analysis would be in terms of f_c and f_p. We could then "plug in" assumptions about f_c and f_p to analyze concrete instances of the general fold function such as insertion-sort. Such assumptions and analyses could be exact or asymptotic, as the application demands. One can certainly imagine that such analyses are likely to be important when reasoning about large modular programs.

4.1.2 Map

Perhaps surprisingly, the analysis of the higher-order map function defined by

```
map = λh.λxs.fold xs of (nil,
        [y, ys, w](h(y) :: w))
```

is more straightforward than for insertion-sort. Again following the approach above and writing $f_{map}(h, xs)$ for $\|\mathtt{map}\| * h * xs$ and $g(z)$ for $\mathtt{pfold}\ z\ \mathtt{of}\ ((1,0), [p, ps, w](4 + (h_p(p))_c, 1 + w_p))$ we have

$$f_{map}(h, xs) = \mathrm{dally}(4 + h_c + xs_c, g(xs_p))$$
$$g_c(0) = 1 \qquad g_c(q+1) = 7 + (h_p(1))_c + g_c(q)$$
$$g_p(0) = 0 \qquad g_p(q+1) = 1 + g_p(q)$$

The cost of applying h to any element of xs is fixed, because the size (potential) of every integer is 1; call this cost C. We conclude that $g(z) \le ((7 + C)z + 1, z)$ and hence $f_{map}(h, xs) \le ((7 + C)xs_p + 5 + h_c + xs_c, xs_p)$.

5. Soundness of the translation

We saw in our examples that the translations of the target-language programs yield expressions that, modulo some manipulations, describe the expected upper bounds on the complexities (and hence evaluation costs) of the original programs. One might worry that the manipulations themselves are the source of success, rather than the translation. Our main goal is to show that (in an appropriate sense), $\mathrm{cost}(t) \le \mathrm{cost}([\![\|t\|]\!])$ for all programs. The challenge in doing so is that cost (and potential) is not compositional: $\mathrm{cost}(rs)$ is not defined solely in terms of the cost of subexpressions of r and s. To get around this, we define a relation between target language closures and complexities, which we then generalize to target and complexity expressions. The relation itself is essentially a logical relation [Mitchell 1996, Ch. 8] that allows us to prove our main Soundness result by induction on terms. With this in mind, we start by defining the *bounding relation* $\sqsubseteq$ as follows. Let t be a target-language expression, ξ a value environment defined on the free variables of t, σ be a target language type, and χ be a complexity such that $\chi \in [\![\|\sigma\|]\!]$. We define $t\,\xi \sqsubseteq_\sigma \chi$ if $t\,\xi \downarrow v\,\theta$ implies

- $\mathrm{cost}(t\,\xi) \le \mathrm{cost}(\chi)$; and
- $v\,\theta \sqsubseteq_\sigma^{\mathrm{val}} \mathrm{pot}(\chi)$.

The *value bounding relation* $\sqsubseteq^{\mathrm{val}}$ relates values to potentials:

- $(tt)\theta \sqsubseteq_{\mathtt{bool}}^{\mathrm{val}} 1$, $(f\!f)\theta \sqsubseteq_{\mathtt{bool}}^{\mathrm{val}} 1$, $n\,\theta \sqsubseteq_{\mathtt{int}}^{\mathrm{val}} 1$.
- $(n_0, \ldots, n_{k-1})\,\theta \sqsubseteq_{\mathtt{int}*}^{\mathrm{val}} p$ if $k \le p$.
- $(\lambda x.r)\theta \sqsubseteq_{\sigma\to\tau}^{\mathrm{val}} p$ if whenever $z\eta \sqsubseteq_\sigma^{\mathrm{val}} q$, then $r\,\theta\{x \mapsto z\eta\} \sqsubseteq_\tau p(q)$.

We will usually drop the type-subscript from $\sqsubseteq$ and $\sqsubseteq^{\mathrm{val}}$. If ξ and ξ^* are value- and complexity- environments respectively, we write $\xi \sqsubseteq \xi^*$ to mean that $x\,\xi \sqsubseteq_\sigma \xi^*(x)$ whenever $x \in \mathrm{Dom}\,\xi^*$ and $\xi^*(x) \in [\![\|\sigma\|]\!]$. If t is a target expression and $\Gamma^* \vdash t^* : \tau$ a complexity term, then we write $t \sqsubseteq \Gamma^* \vdash t^* : \tau$ to mean that $t\,\xi \sqsubseteq [\![t^*]\!]\xi^*$ whenever ξ^* is a Γ^*-consistent environment and $\xi \sqsubseteq \xi^*$.

Our main theorem is the following:

THEOREM 2 (Soundness). *If $\Gamma \vdash t : \tau$, then $t \sqsubseteq \|\Gamma \vdash t : \tau\|$.*

COROLLARY 3. *If $\_ \vdash t : \tau$, then $\mathrm{cost}(t\,\{\}) \leq \mathrm{cost}(\|\_ \vdash t : \tau\|)$.*

The reader may be concerned by the seeming lack of a connection between any putative value environment ξ under which t may be evaluated and Γ; in particular, there is no connection between $\xi(x)$ and $\Gamma(x)$ for $x \in \mathrm{fv}(t)$. A short response is that the Soundness Theorem is typically applied only to closed terms (as in the Corollary) and hence there is no concern, and the reader is free to accept this and ignore the remainder of this paragraph. A longer response is that this observation has to do with the fact that our evaluation semantics places no restrictions upon the value environment to enforce reasonable "typing." Indeed, no restrictions can be placed, as we have not defined a notion of typing for values, although defining such a notion is not technically difficult. However, in applying the Soundness Theorem and chasing the definitions, one is forced to apply it to value environments that assign "reasonable" values to the variables. For suppose that $\Gamma \vdash t : \tau$; $t \sqsubseteq \|\Gamma \vdash t : \tau\|$; ξ is a value environment such that $t\,\xi \downarrow v\,\theta$; and that $x \in \mathrm{fv}(t)$. Since $x \in \mathrm{fv}(t)$, $x \in \mathrm{Dom}\,\Gamma$. Take any $\|\Gamma\|$-consistent environment ξ^* such that $\xi \sqsubseteq \xi^*$. Suppose that $\xi^*(x) \in [\![\|\sigma\|]\!]$; by the definition of consistency, $\|\Gamma\|(x) = \|\sigma\|$ and hence $\Gamma(x) = \sigma$. But in addition, since $x\,\xi \sqsubseteq_\sigma \xi^*(x)$ and $x\,\xi \downarrow \xi(x)$, $\xi(x) \sqsubseteq^{\mathrm{val}}_\sigma \xi^*(x)$. Examining the definition of $\sqsubseteq^{\mathrm{val}}_\sigma$, we see that this ensures that the "type" of $\xi(x)$ (had we defined it) must be σ.

Before proving the Soundness Theorem, we establish some preliminary lemmas. The first three are consequences of definitions and syntactic manipulation.

LEMMA 4. *If $t\,\xi \sqsubseteq_\tau \chi$ then for all $\chi' \in [\![\|\tau\|]\!]$, $t\,\xi \sqsubseteq \chi \vee \chi'$.*

LEMMA 5. *If $\xi \sqsubseteq \xi^*$ and $v\,\theta \sqsubseteq^{\mathrm{val}} q$, then $\xi\{x \mapsto v\,\theta\} \sqsubseteq \xi^*\{x \mapsto (1,q)\}$.*

LEMMA 6. *$[\![t]\!]\xi\{x \mapsto (a,b)\} = [\![t\{x \mapsto (a,y)\}]\!]\xi\{y \mapsto b\}$ where y is a fresh variable.*

The next two lemmas establish bounds related to recursive definitions.

LEMMA 7. *For all complexity expressions r,*

$$\mathrm{cost}([\![s]\!]\xi^*) \leq \mathrm{cost}([\![\mathtt{pfold}\ r\ \mathtt{of}\ (s,[p,ps,w]t)]\!]\xi^*).$$

Proof. Formally we prove by induction on q that for all complexity expressions r and environments ξ^*, if $[\![r]\!]\xi^* = q$, then

$$\mathrm{cost}([\![s]\!]\xi^*) \leq \mathrm{cost}([\![\mathtt{pfold}\ r\ \mathtt{of}\ (s,[p,ps,w]t)]\!]\xi^*).$$

If $q = 0$, then the two sides of the inequality are in fact equal. Suppose $[\![r]\!]\xi^* = q+1$. Then

$$\mathrm{cost}([\![\mathtt{pfold}\ r\ \mathtt{of}\ (s,[p,ps,w]t)]\!]\xi^*) = 2 + \mathrm{cost}(rec) + \mathrm{cost}([\![t]\!]\xi^*\{p,ps,w \mapsto 1,q,(1,\mathrm{pot}(rec))\})$$

where $rec = [\![\mathtt{pfold}\ y\ \mathtt{of}\ (s,[p,ps,w]t)]\!]\xi^*\{y \mapsto q\}$. By the induction hypothesis, $\mathrm{cost}([\![s]\!]\xi^*) \leq \mathrm{cost}(rec)$, and so the claim follows. □

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

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

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

Proof. We prove the claim by induction on k. If $k = 0$ then the cost of the fold expression is $2 + \mathrm{cost}(s\xi_0)$, and the value of the fold expression under ξ_0 is the value to which $s\xi_0$ evaluates. The cost bound is proved by Lemma 7:

$$\mathrm{cost}(\mathtt{fold}\ y\ \mathtt{of}\ (s,[x,xs,w]t)\,\xi_0) \leq 2 + (\|s\|\xi_0^*)_c \leq 2 + ([\![\mathtt{pfold}\ y\ \mathtt{of}\ (\|s\|,[p,ps,w]t)]\!]\xi_0^*)_c$$

By the induction hypothesis $v\,\theta \sqsubseteq^{\mathrm{val}} (\|s\|\xi_0^*)_p$. Since the potential of $\mathrm{dally}(2,[\![\mathtt{pfold}\ y\ \mathtt{of}\ (\|s\|,[p,ps,w]t)]\!]\xi_0^*)$ is either $(\|s\|\xi_0^*)_p$ or $(\|s\|\xi_0^*)_p \vee ([\![t']\!]\xi_0^{**})_p$ for some environment ξ_0^{**}, we know from Lemma 4 that the potential bound holds.

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

$$\mathtt{fold}\ y\ \mathtt{of}\ (s,[x,xs,w]t)\,\xi\{y \mapsto ns\} \sqsubseteq \mathrm{dally}(2, rec)$$

where $rec = [\![\mathtt{pfold}\ y\ \mathtt{of}\ (\|s\|,[p,ps,w]t)]\!]\xi^*\{y \mapsto q\}$. So if

$$(\mathtt{fold}\ y\ \mathtt{of}\ (s,[x,xs,w]t))\xi\{y \mapsto ns\} \downarrow v'\,\theta'$$

then $v'\,\theta' \sqsubseteq^{\mathrm{val}} rec_p$, which means

$$\xi\{x,xs,w \mapsto n,ns,v'\theta'\} \sqsubseteq \xi^*\{x,xs,w \mapsto (1,1),(1,q),(1,rec_p)\}$$

Let $\xi_1 = \xi_0\{x,xs,w \mapsto n,ns,v'\theta'\}$ and let $\xi_1^* = \xi_0^*\{p,ps,w \mapsto 1,q,(1,rec_p)\}$. Since y does not occur free in t or t', by Lemma 6, $t\,\xi_1 \sqsubseteq [\![t']\!]\xi_1^*$. So

$$\begin{aligned}
&\mathrm{cost}(\mathtt{fold}\ y\ \mathtt{of}\ (s,[x,xs,w]t)\,\xi_0)\\
&\quad = 2 + \mathrm{cost}(\mathtt{fold}\ y\ \mathtt{of}\ (s,[x,xs,w]t)\,\xi\{y \mapsto ns\})\\
&\qquad + \mathrm{cost}(t\,\xi_1)\\
&\quad \leq 2 + (2 + rec_c) + \left([\![t']\!]\xi_1^*\right)_c\\
&\quad = 2 + ([\![\mathtt{pfold}\ y\ \mathtt{of}\ (\|s\|,[p,ps,w]t)]\!]\xi_0^*)_c
\end{aligned}$$

For the potential bound, if $(\mathtt{fold}\ y\ \mathtt{of}\ (s,[x,xs,w]t))\xi_0 \downarrow v\,\theta$, then $t\,\xi_1 \downarrow v\,\theta$, so $v\,\theta \sqsubseteq^{\mathrm{val}} \mathrm{pot}([\![t']\!]\xi_1^*)$. By Lemma 4, we have

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

□

Proof of Soundness Theorem. By induction on the derivation of $\Gamma \vdash t : \tau$. Most of the proof is similar to the analogous proof in [Danner and Royer 2009], so we do just a few cases here. In this proof we will write $\|s\|\xi^*$ for $[\![\|s\|]\!]\xi^*$.

Suppose $\Gamma \vdash (r :: s) : \mathtt{int}*$ and fix $\xi \sqsubseteq \xi^*$. We have that $\|r :: s\| = (1 + \|r\|_c + \|s\|_c, 1 + \|s\|_p)$. For the cost bound,

$$\mathrm{cost}((r :: s)\xi) = 1 + \mathrm{cost}(r\,\xi) + \mathrm{cost}(s\,\xi) \leq 1 + \mathrm{cost}(\|r\|\xi^*) + \mathrm{cost}(\|s\|\xi^*) = \mathrm{cost}(\|r :: s\|\xi^*).$$

For the potential bound, if $(r :: s)\xi \downarrow (n,n_0,\ldots,n_{k-1})$, then $s\,\xi \downarrow (n_0,\ldots,n_{k-1})$, so by the induction hypothesis, $\mathrm{pot}(\|s\|\xi^*) = q$ for some $q \geq k$; hence $\mathrm{pot}(\|r :: s\|\xi^*) = 1 + q \geq 1 + k$.

Suppose $\Gamma \vdash \lambda x.r : \sigma \to \tau$, with $\Gamma, x \mapsto \sigma \vdash r : \tau$, and fix $\xi \sqsubseteq \xi^*$. Verifying the cost bound is trivial, so we focus on the potential bound. Set $p = \|\lambda x.r\|_p\xi^* = \lambda\!\!\lambda v.\|r\|\xi^*[x \mapsto (1,v)]$. We must show that if $z\,\eta \sqsubseteq^{\mathrm{val}}_\sigma q$, then $r\,\xi\{x \mapsto z\,\eta\} \sqsubseteq_\tau p(q) = \|r\|\xi^*\{x \mapsto (1,q)\}$. Since $z\,\eta \sqsubseteq^{\mathrm{val}} q$, by Lemma 5 $\xi\{x \mapsto z\,\eta\} \sqsubseteq \xi^*\{x \mapsto (1,q)\}$, and so the claim follows by the induction hypothesis for r.

Suppose $\Gamma \vdash rs : \tau$, with $\Gamma \vdash r : \sigma \to \tau$ and $\Gamma \vdash s : \sigma$, and fix $\xi \sqsubseteq \xi^*$. By unraveling definitions,

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

Suppose $(rs)\xi \downarrow v\,\theta$ by the following evaluation rule:

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

We first show that

$$r'\,\theta_1\{x \mapsto w\,\theta_2\} \sqsubseteq (\|r\|\xi^*)_p(\|s\|\xi^*)_p. \qquad (*)$$

Since $r\,\xi \downarrow (\lambda x.r')\theta_1$, we know that $(\lambda x.r')\theta_1 \sqsubseteq^{\mathrm{val}} \mathrm{pot}\|r\|\xi^*$. Hence if $z\,\eta \sqsubseteq^{\mathrm{val}} p$ then $r'\,\theta_1\{x \mapsto z\,\eta\} \sqsubseteq (\|r\|\xi^*)_p(p)$. Since $s\,\xi \sqsubseteq \|s\|\,\xi^*$ and $s\,\xi \downarrow w\,\theta_2$, we have that $w\,\theta_2 \sqsubseteq^{\mathrm{val}} (\|s\|\xi^*)_p$, and $(*)$ follows.

To establish the cost bound, we compute

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

with the inequality following from the induction hypotheses and $(*)$. The potential bound follows from $(*)$ and the fact that $r'\,\theta_1[x \mapsto w\,\theta_2] \downarrow v\,\theta$.

Suppose $\Gamma \vdash \mathtt{fold}\,r\,\mathtt{of}\,(s,[x,xs,w]t) : \tau$ and set $t' = \|t\|\{x, xs \mapsto (1,p),(1,ps)\}$ so that

$$\|\mathtt{fold}\,r\,\mathtt{of}\,(s,[x,xs,w]t)\| = \mathtt{pfold}\,\|r\|_p\,\mathtt{of}\,(\|s\|,[p,ps,w]t)'.$$

Suppose $r\,\xi \downarrow ns$; we prove the claim by induction on ns. Suppose $ns = ()$. We start by computing the cost:

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

The second inequality is an equality if $\|r\|_p = 0$, or follows from Lemma 7 if $\|r\|_p > 0$. Turning to the potential bound, if $(\mathtt{fold}\,r\,\mathtt{of}\,(s,[x,xs,w]t))\xi \downarrow v\,\theta$ then $s\,\xi \downarrow v\,\theta$ as well and so $v\,\theta \sqsubseteq^{\mathrm{val}} \mathrm{pot}(\|s\|\xi^*)$. If $\|r\|_p = 0$, this suffices to verify the claim. If $\|r\|_p > 0$, use Lemma 4. This finishes the case in which $r\,\xi \downarrow ()$.

Suppose $r\,\xi \downarrow (n,ns)$, where $ns = (n_0,\dots,n_{k-1})$. By the induction hypothesis for r, $\|r\|_p\xi^* = q+1$ for some $q \ge k$. Make the following definitions:

- $\xi_0 = \xi\{y \mapsto ns\}$; $\xi_0^* = \xi^*\{y \mapsto q\}$.
- $rec^t = \mathtt{fold}\,y\,\mathtt{of}\,(s,[x,xs,w]t)$.
- $rec = [\![\mathtt{pfold}\,y\,\mathtt{of}\,(\|s\|,[p,ps,w]t')]\!]\xi_0^*$.

By Lemma 8, $rec^t\,\xi_0 \sqsubseteq \mathrm{dally}(2, rec)$. So if $rec^t\,\xi_0 \downarrow v'\,\theta'$ then $v'\,\theta' \sqsubseteq^{\mathrm{val}} rec_p$. Now set

- $\xi_1 = \xi\{x,xs,w \mapsto n,ns,v'\theta'\}$.
- $\xi_1^* = \xi^*\{p,ps,w \mapsto 1,q,(1,rec_p)\}$.

By Lemma 5 $\xi_1 \sqsubseteq \xi_1^*$, so by the induction hypothesis for t and Lemma 6, we have $t\,\xi_1 \sqsubseteq [\![t']\!]\xi_1^*$.

For the cost bound when $r\,\xi \downarrow (n,ns)$, we have

$$\begin{aligned}&\mathrm{cost}(\mathtt{fold}\,r\,\mathtt{of}\,(s,[x,xs,w]t)\,\xi)\\ &\quad = 1 + \mathrm{cost}(r\,\xi) + \mathrm{cost}(rec^t\,\xi\{y \mapsto ns\}) + \mathrm{cost}(t\,\xi_1)\\ &\quad \le 1 + (\|r\|\xi^*)_c + (2 + rec_c) + ([\![t']\!]\xi_1^*)_c\\ &\quad = 1 + (\|r\|\xi^*)_c +\\ &\qquad\quad \mathrm{cost}([\![\mathtt{pfold}\,\|r\|_p\,\mathtt{of}\,(\|s\|,[p,ps,w]t')]\!]\xi^*)\\ &\quad = \mathrm{cost}(\|\mathtt{fold}\,r\,\mathtt{of}\,(s,[x,xs,w]t)\|\xi^*)\end{aligned}$$

For the potential bound, suppose $(\mathtt{fold}\,r\,\mathtt{of}\,(s,[x,xs,w]t))\xi \downarrow v\,\theta$. Then we must have $t\,\xi_1 \downarrow v\,\theta$ as well; hence $v\,\theta \sqsubseteq^{\mathrm{val}} \mathrm{pot}([\![t']\!]\xi_1^*)$. So by (4),

$$\begin{aligned}v\,\theta \sqsubseteq^{\mathrm{val}} \mathrm{pot}(\|s\|\xi^*) \vee \mathrm{pot}([\![t']\!]\xi_1^*) &=\\ \mathrm{pot}(\|\mathtt{fold}\,r\,\mathtt{of}\,(s,[x,xs,w]t)\|\xi^*).\ \square\end{aligned}$$

6. Implementation in Coq

We have implemented a subset of the target and complexity languages in Coq that includes the simply typed λ-calculus with integer and boolean operations. For this subset we have implemented the translation function and proof of the Soundness Theorem. The current development may be found at

`http://wesscholar.wesleyan.edu/compfacpub`.

When complete, the formalization will provide a mechanism for certified upper bounds on the cost of programs in the target language. Since the denotational semantics is built on Coq's built-in type system, one can use the formalized Soundness Theorem to establish a bound on a given target program, then continue to reason in Coq to simplify the bound, for example establishing a closed form if desired.

Although we use Coq to formalize the translation, our system is really a blend of external and internal verification. Our long-term goal is to be able to translate from programs written in a language such as SML or OCaml. Such programs would not have cost information directly associated with them via annotations (such as done by Danielsson [2003]) or a type system. In this sense, our approach follows that of CFML [Charguéraud 2010], in which Caml source code is translated into a formula that can be used to verify post-conditions.

The main non-trivial aspect of the development is the definition of the bounding relation. The bounding relation is a simultaneous recursive definition to two relations, $\sqsubseteq$ and $\sqsubseteq^{\mathrm{val}}$. In Coq the difficulty arises in defining $\sqsubseteq_{\sigma\to\tau}$ in terms of $\sqsubseteq^{\mathrm{val}}_{\sigma\to\tau}$ (in turn defined in terms of $\sqsubseteq_\tau$), which is not a structural descent on type. We resolve this by defining subsidiary versions of $\sqsubseteq$ and $\sqsubseteq^{\mathrm{val}}$ that take a natural number argument, and which are structurally decreasing on that argument. $\sqsubseteq$ and $\sqsubseteq^{\mathrm{val}}$ are then defined in terms of these relations, using a numeric value sufficiently high that the inductive definition is guaranteed to terminate by reaching a base type, rather than a value of 0 for the numeric argument.[7] With this out of the way, the proof of the Soundness Theorem proceeds more-or-less as described in this paper.

7. Related work

The core idea of this paper is not new. There is a reasonably extensive literature over the last several decades on (semi-)automatically constructing resource bounds from source code. The first work naturally concerns itself with first-order programs. Wegbreit [1975] describes a system for analyzing simple Lisp programs that produces closed forms that bound running time. An interesting aspect of this system is that it is possible to describe probability distributions on the input domain (e.g., the probability that the head of an input list will be some specified value), and the generated bounds incorporate this information. Rosendahl [1989] proposes a system based on step-counting functions and abstract interpretation for a first-order subset of Lisp. More recently the COSTA project (see, e.g., Albert et al.

[7] We also implemented a version in which $\sqsubseteq^{\mathrm{val}}$ is inlined into the definition of $\sqsubseteq$ as an anonymous fixpoint, but found that it became even more tedious to prove the required lemmas, because we often have to reason specifically about $\sqsubseteq^{\mathrm{val}}$.

[2007]) has focused on automatically computing resource bounds for imperative languages (actually, bytecode).

Le Métayer's [1988] ACE system is a two-stage system that first converts FP[8] programs to recursive FP programs describing the number of recursive calls of the target program, then attempts to transform the result using various program-transformation techniques to obtain a closed form. Jost et al. [2010] describe a formalism for automatically infering linear resource bounds on higher-order programs, provided of course that such bounds are correct, and Hoffmann and Hofmann [2010] extend this work to handle polynomial bounds. This system involves a type system analogous to the target-language system, but in which the types are annotated with variables corresponding to resource usage. Type inference in the annotated system comes down to solving a set of constraints among these variables.

Here let us delve a little more into the systems mentioned in the introduction, and just those aspects concerned with a general analysis of higher-type call-by-value languages. Shultis [1985] defines a denotational semantics for a simple higher-order language that models both the value and the cost of an expression. As a part of the cost model, he develops a system of "tolls," which play a role similar to the potentials we define in our work. The tolls and the semantics are not used directly in calculations, but rather as components in a logic for reasoning about them. Sands [1990] puts forward a translation scheme in which programs in a target language are translated into programs *in the same language* that incorporate cost information; several target languages are discussed, including a higher-order call-by-value language. Each identifier f in the target language is associated to a *cost closure* that incorporates information about the value f takes on its arguments; the cost of applying f to arguments; and arity. Cost closures are intended to address the same issue our higher-type potentials do: recording information about the future cost of a partially-applied function. Van Stone [2003] annotates the operational semantics for a higher-order language with cost information. She then defines a category-theoretic denotational semantics that uses "cost structures" (which are related to monads) to capture cost information and shows that the latter is sound with respect to the former. Benzinger [2004] annotates NuPRL's call-by-name operational semantics with complexity estimates. The language for the annotations is left somewhat open so as to allow greater flexibility. The analysis of the costs is then completed using a combination of NuPRL's proof generation and Mathematica. Benzinger's is the only system to explicitly involve automated theorem proving, though Sand's could also do so.

These last formalisms are closest to ours in approach, but differ from ours in a key respect: the cost domain incorporates information about values in the target domain so as to provide exact costs, whereas our approach focuses on upper bounds on costs in terms of input size. We are hopeful that our system proves amenable to analyzing complex programs, but there is much work yet to be done.

8. Conclusions and further work

We have described a static complexity analysis for a higher-order language with structural list recursion that yields an upper bound on the evaluation cost of any typeable program in the target language. It proceeds by translating each target-language program t into a program $\|t\|$ in a complexity language. We prove a Soundness Theorem for the translation that has as a consequence that the cost component of $\|t\|$ is an upper bound on the evaluation cost of t. By formalizing the translation and proof of the Soundness Theorem in Coq, we obtain a machine-checkable certification of that upper bound on evaluation cost.

The language described here supports only structural recursion on lists; an obvious extension would be to handle general recursion. This should be straightforward if we require the user to supply a proof of termination of the program to be analyzed. However, it should be possible to define the operational semantics of the target language co-inductively (as done by, e.g., [Leroy and Grall 2009]), thereby allowing explicitly for non-terminating computations. The complexity language semantics would then have to be adapted so that the denotation of a recursive complexity function may be partial; the foundations for such denotational semantics have already been carried out by Paulin-Mohring [2009] and Benton et al. [2009]. Indeed, one could then hope to prove termination by extracting complexity bounds and then proving that these bounds in fact define total functions.

Although we have adapted our formalism to a few other inductively-defined datatypes (e.g., binary trees), we suspect that there are hidden difficulties in generalizing the system to handle arbitrary inductive definitions. One such difficulty might be, e.g., binary trees in which the external nodes are labeled by values of another inductively-defined datatype. If we wish to consider both trees and labels as contributing to size (potential), then it seems that only external nodes labeled by size-0 values have potential 0. That in turn may make it difficult to develop useful cost bounds for functions that ignore the labels. Jost et al. [2010] deal with this issue by annotating the type constructors with resource information, and hence automatically account for the resource information for all types of objects reachable from the root of a given value. We have investigated a similar approach in our setting, in which potential types mirror the target language types. For example, we would define $\langle\langle \sigma\,\mathtt{list}\rangle\rangle$ to be essentially $\langle\langle \sigma\rangle\rangle\ \mathtt{list}$. This becomes rather burdensome in practice, and a mechanism for minimizing the overhead when the generality is not desired would be a necessity.

Another obvious direction would be to handle different evaluation strategies and notions of cost. Compositionality is a thorny issue when considering call-by-need evaluation and lazy datatypes, and it may be that amortized cost is at least as interesting as worst-case cost. Sands [1990], Van Stone [2003], and Danielsson [2003] address laziness in their work. The call-by-push-value paradigm [Levy 1999] gives an alternative perspective on our complexity analysis. Call-by-push-value disinguishes values from computations in a monadic-like approach under the maxim "a value is, a computation does." With this in mind, we might adopt the following statement with respect to complexities: "potential measures what is, cost measures what happens." An alternative presentation of our work might utilize a call-by-push-value target language to emphasize the distinction between computation expressions and value expressions and what those mean for the complexity analysis.

The development in Coq currently works directly with the expressions of the form $\|t\|$. These are moderately messy, as can be seen from the examples. It would be nice to provide a more elegant presentation of these complexities along the lines as the discussion of the examples. In the same vein, it would be very useful to develop tactics that allow users to transform complexities into simpler ones, all the while ensuring the appropriate bounds still hold. Benzinger [2004] addresses this idea, as do Albert et al. [2011] of the COSTA project. Another relevant aspect of the COSTA work is that their cost relations use non-determinism where we have used a maximization operation to handle conditional constructs; it would be very interesting to see how their approaches play out in our context. These tactics could be then applied manually or automatically according to the user's preferences using Coq's built in automation tools. Ultimately we should have a library of tactics for transforming the recurrences produced by the translation function to closed (possibly asymptotic) forms when possible.

[8] I.e., Backus' language FP [Backus 1978].

Acknowledgments

We would like to thank the referees for several helpful suggestions including pointers to additional related work.

References

E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In R. D. Nicola, editor, *Programming Languages and Systems: 16th European Symposium on Programming, ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, 2007. doi: 10.1007/978-3-540-71316-6_12.

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

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

N. Benton, A. Kennedy, and C. Varming. Some domain theory and denotational semantics in Coq. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 115–130. Springer-Verlag, 2009. doi: 10.1007/978-3-642-03359-9_10.

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

A. Charguéraud. Program verification through characteristic formulae. In P. Hudak and S. Weirich, editors, *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 321–332. ACM Press, 2010. doi: 10.1145/1863543.1863590.

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

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

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

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

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

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

P. Levy. Call-by-push-value: A subsuming paradigm. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99*, volume 1581 of *Lecture Notes in Computer Science*, pages 644–644. Springer-Verlag, 1999. doi: 10.1007/3-540-48959-2_17.

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

C. Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin, editors, *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, pages 383–413. Cambridge University Press, 2009. doi: 10.1017/CBO9780511770524.018.

J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

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

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

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

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

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