Automated higher-order complexity analysis

Ralph Benzinger

Department of Computer Science, Cornell University, Ithaca, NY 14853, USA

Abstract

This paper describes the automated complexity analysis (ACA) system for automated higher-order complexity analysis of functional programs synthesized with the NUPRL proof development system. We introduce a general framework for defining models of computational complexity for functional programs based on an annotation of a given operational language semantics. Within this framework, we use type decomposition and polynomialization to express the complexity of higher-order terms. Symbolic interpretation of open terms automates complexity analysis, which involves generating and solving higher-order recurrence equations. Finally, the use of the ACA system is demonstrated by analyzing three different implementations of the pigeonhole principle. © 2003 Elsevier B.V. All rights reserved.

Keywords: Computational complexity analysis; Functional programs; Higher-order complexity; Program synthesis; Feasible mathematics

1. Introduction—the pigeonhole incident

In a memorable episode during the late 1990s, the NUPRL research group at Cornell University was collectively trying to locate the source of the exponential running time exhibited by a provably correct state-minimization algorithm synthesized with the NUPRL proof development system [14]. After a week-long manual search through the system’s proof library, the problem was eventually traced down to a proof of the pigeonhole principle, which was cited as a lemma in the main theorem. Once the exponential running time had been linked to the pigeonhole lemma, the state-minimization algorithm was easily made polynomial by revising the proof of the lemma [21].

1.1. The missing link in program synthesis

The pigeonhole incident, as this little episode came to be known within the NUPRL group, illustrates the missing link in machine-assisted program synthesis from formal
specifications. Current automated synthesis tools are too narrowly focused on program correctness and automation, while paying little attention to the efficiency of the generated code. Some recent systems acknowledge this deficiency by granting the user extended control over algorithm design, but as the pigeonhole incident aptly showed, even experienced users may have difficulties in generating efficient programs.

There is a growing literature on feasible mathematics that addresses this specification gap. The key insight is to adopt a computational viewpoint by acknowledging that functions differ in their “intrinsic difficulty” [3]. Of particular interest is the class of feasible functions that are “amenable” to computation, which is usually meant to refer to the class of polynomial-time functions. The traditional approach to feasible mathematics is to limit the specification language so that only efficient programs can be defined. This line of research has led to a number of logical theories of restricted expressiveness, in which infeasible functions cannot exist [9,15,17,20]. An alternative approach to feasibility is to define some notion of computational complexity within the language, e.g., by reflection [1,2,11]. Ideally, any such theory should integrate complexity judgments seamlessly into the language by hiding the evaluation details of the underlying low-level machine models. It would be unrealistic, however, to expect that we can get this integration for free—in the end, the greater flexibility in deriving a particular algorithm is offset by the additional obligation to prove its actual feasibility.

The present work automates the construction of this proof, thus eliminating the major disadvantage of the reflection method for feasible program synthesis. The underlying theoretical framework allows the definition of general models of higher-order computational complexity and applies to any formal system based on operational semantics that is able to express evaluation in its term language. The ACA system introduced in this paper is a reference implementation of these concepts that automatically computes and proves correct a closed upper bound on the worst-case time complexity of functional programs generated by the Nuprl system. Among the examples successfully analyzed by the ACA system are several algorithms extracted from different proofs of the pigeonhole principle.

### 1.2. The ACA system

The Nuprl proof development system [13] is an environment for machine-assisted proof and program development based on an extension of Martin-Löf type theory [12,18]. One of its distinguishing features is the realization of the proofs-as-programs principle [3] that facilitates the extraction and execution of computational content from formal proofs constructed by the system. Given a type-theoretic proof of some sequent ⊢ P, Nuprl automatically generates a functional program p of type [P] that by design is guaranteed to be correct with respect to P. As an example, a constructive proof of

\[ \forall x : \mathbb{N}. \exists y : \mathbb{N}. y^2 \leq x < (y + 1)^2 \]

would yield a function f that computes the integer square root for any natural number x. While all functions extracted from different proofs of above theorem are extensionally equal, their efficiency will depend non-trivially on the proof structure. The interested
The ACA system developed by the author automatically analyzes the computational complexity of NUPRL proof extracts or other functional programs written in NUPRL’s term language. It interfaces with the MATHEMATICA computer algebra system through a syntactic term translator to perform arithmetical simplifications and to solve recurrence equations. The main ACA modules are the symbolic evaluator, written in NUPRL ML, the recurrence solver, written in MATHEMATICA’s functional programming language, and the proof generator, written in NUPRL ML and complemented by a NUPRL theory containing miscellaneous definitions and theorems.

The user interacts with the ACA system by issuing text commands at the NUPRL top-level prompt M> to generate, query, and manipulate the recurrence equations for the term being analyzed. The main analysis function cc expects a symbolic term $t$ with free variables and returns a closed expression for the complexity of $t$ in terms of the free variables of $t$, where $v_-$ denotes the complexity of a term substituted for variable $v$:

\[
M> \text{cc } \text{ind}(m; a; i.z.i+z) ;
1 + a_- + m_- + 2 * m : \text{term}
\]

The system also computes a proof term $pf$ asserting the complexity of $t$ and a proof tactic $tac$ that will generate a NUPRL proof tree for $\vdash pf$. The details of this construction are beyond the scope of this paper but can be found in [6].

1.3. Previous work

Automated complexity analysis is a perennial yet surprisingly disregarded aspect of static program analysis that dates back to the 1970s. The seminal contribution to this area was Wegbreit’s METRIC system [29] for analyzing simple programs written in a first-order subset of LISP. The system returns an upper bound on the worst-case complexity, a lower bound on the best-case complexity, and the expected average-case complexity, although the latter requires the user to supply probability distributions for all conditional tests that occur in the program. The ACE system [19] by Le Métayer uses a library of transformation rules to translate a program $P$ written in the FP language into a step-counting version $P'$ that for each argument $\text{LDY+}$ returns the number of primitive reduction steps $P'(\text{LDY+})$ required for computing $P(\text{LDY+})$. The program $P'$ is then transformed into a composition of basic functions whose computational behavior is known. Rosendahl [22,23] implemented a system for automatic complexity analysis of first-order LISP programs that takes a program $P$ and generates a step-counting version $P'$. From this program $P'$, a partial time bound function $t_P$ is derived through abstract interpretation of $P'$, so that $t_P$ returns an upper bound on the number of reduction steps for any input of a given size.

Theoretical advances for analyzing lazy functional languages were made by Wadler [28] and Bjerner and Holmström [8], who used projections and demand analysis to model an informal call-by-need reduction strategy for the untyped lambda calculus. Sands [24,25] extends the technique of cost counting programs described in [19,22,23]...
to higher-order programs, where his cost closures retain the evaluation history of higher-order functions. His method is able to translate programs into recursive higher-order equations but does not provide any means for solving these equations. The author’s previous work on automated complexity analysis [7] describes the ACA prototype for computing upper bounds on the time complexity of first-order Nuprl programs. The implementation uses symbolic evaluation of open terms to derive recursive cost equations, which are solved by Mathematica. The current paper can be seen as a direct extension of this work to higher-order complexity.

1.4. How to read this paper

We feel that some comments might be helpful to readers who are unfamiliar with the Nuprl system. Terms and term variables are alphanumeric sequences including %, @, and _ symbols and are set in typewriter font. We generally state terms literally but may revert to their mathematical display form if we think it benefits readability; in particular, we write \( \lambda x.b \) and \( f \, x \) instead of \( \text{lambda}(x.b) \) and \( \text{apply}(f; x) \), respectively. Application binds stronger than abstraction, so \( \lambda x.f \, t \) stands for \( \lambda x.\text{apply}(f; t) \). The inductive term \( \text{ind}(m; b; i,z,s) \) encodes primitive recursion on \( m \) with base case \( b \) and step case \( i,z,s \). In a recursive call, the binding variables \( i \) and \( z \) occurring freely in \( s \) are substituted with the value of \( m \) and the result of the recursive call \( \text{ind}(m-1; b; i,z,s) \), respectively. Mathematical variables denoting terms or values are set in italic font. We write \( \langle x \rangle \) for a list of values \( \langle x_1, \ldots, x_n \rangle \) and use the symbol \( \oplus \) for list concatenation. The set of integers \( \{0,1,\ldots,k-1\} \) is abbreviated to \( \mathbb{N}_k \), and \( [P] \) is shorthand for if \( P \) then 1 else 0 fi.

This paper strives for a natural balance between formalism and intuitionism. When reasoning informally about the evaluation of integer terms, we frequently do not distinguish between a canonical term and its mathematical integer value if the meaning is clear from the context. For example, we may write \( 3*2*1 \downarrow 3! \) in addition to \( 3*2*1 \downarrow 6 \), meaning that \( 3*2*1 \) evaluates to the canonical term whose mathematical value is equal to 3!.

The remainder of the paper is organized as follows: In Section 2, we develop our calculus for reasoning about the computational complexity of higher-order functional programs. Section 3 shows how this calculus is automated within the ACA system for computing upper bounds on the time complexity of Nuprl programs with respect to call-by-name evaluation. Section 4 outlines our approach to generating and solving higher-order recurrence equations that describe the complexity of recursive programs. Finally, Section 5 illustrates the use of the ACA system by analyzing three different versions of the pigeonhole principle.

2. The computational complexity of functional programs

Most formal definitions of computational complexity are based on the Turing machine or the random access machine model and assign certain costs to a
For modern high-level languages featuring advanced data structures or garbage collection, however, the definition of an implementation-independent cost model that accurately reflects the resource usage during program execution becomes increasingly difficult, if not impossible. This principal dilemma between generality and faithfulness is exacerbated by functional programming languages, whose hidden evaluator cost may be significant [16].

In this paper we propose a general framework for reasoning about the computational complexity of functional programs relative to an operational semantics $O$. To define a complexity measure with respect to $O$, we annotate each rule $t_1 \downarrow t_2$ in $O$ with some complexity information $n$, to obtain an annotated semantics $A$. The expression $n$ need not be a term of the language $O$, and there are no a priori restrictions imposed on the syntax of $n$. If we can deduce $t \downarrow^A w$ (in $n$) and $w$ is canonical, we say that $t$ has value $w$ and complexity $n$ relative to $A$, also denoted by $t^*$ and $\hat{t}$, respectively. For brevity, we will write $t \downarrow w$ (in $n$) and $t \downarrow^A$ (in $n$) if $t \downarrow w$ (in $n$) for some (potentially unknown) canonical term $w$.

The semantic interpretation of the complexity annotations $n$ is implicit in $A$. Depending on the computational resource of interest and our assumptions about the underlying machine model, the annotations might model upper bounds, lower bounds, or exact quantities. Consider, for example, the multiplication rule

\[
\frac{u \downarrow k_1, \ v \downarrow k_2}{\text{mul}(u; v) \downarrow k_1 \cdot k_2}
\]
as given by NUPRL’s standard call-by-name semantics $N$ [12, pp. 97–101]. We can model time complexity by defining

\[
\frac{u \downarrow k_1 \ (\text{in} \ n_1), \ v \downarrow k_2 \ (\text{in} \ n_2)}{\text{mul}(u; v) \downarrow k_1 \cdot k_2 \ (\text{in} \ n_1 + n_2 + 1)},
\]
or space complexity by defining

\[
\frac{u \downarrow k_1 \ (\text{in} \ n_1), \ v \downarrow k_2 \ (\text{in} \ n_2)}{\text{mul}(u; v) \downarrow k_1 \cdot k_2 \ (\text{in} \ n_1 + n_2)},
\]
or a combined “space–time” complexity by

\[
\frac{u \downarrow k_1 \ (\text{in} \ n_1), \ v \downarrow k_2 \ (\text{in} \ n_2)}{\text{mul}(u; v) \downarrow k_1 k_2 \ (\text{in} \ n_1 + n_2 + \log(k_1 k_2))}.
\]

Assuming pure call-by-name evaluation, above annotation for time complexity models the exact number of reduction steps as specified by $N$. If we further assume that NUPRL’s term evaluator is a canonical implementation of $N$, we may also infer that our annotation for space complexity yields an upper bound on the maximum size of a term during evaluation.
The ACA system is based on the annotated semantics $\mathcal{F}$ shown in Fig. 1 that models the exact time complexity of programs interpreted by NUPRL’s standard call-by-name evaluator. For simplicity, $\mathcal{F}$ assigns unit cost to each reduction step, but more faithful models using explicit substitution or parameterized annotations are likewise possible. An alternative call-by-value evaluation strategy has already been formalized and can be found in [6].

Given an annotated semantics $\mathcal{A}$, we can compute the complexity relative to $\mathcal{A}$ of any closed term by reducing it to canonical form. Given, for example, the Fibonacci function

\[
\text{fibo} == \lambda n.\text{ind}(n; \lambda x,y.\lambda i,z.\lambda x,y,z. y ((x+y) 0 1)),
\]

Fig. 1. Annotated semantics $\mathcal{F}$ for NUPRL’s call-by-name evaluator.
the cost of computing the tenth Fibonacci number $F_{10}$ using \texttt{fibo} is 123 relative to $T$, since $\texttt{fibo}\ 10 \downarrow \texttt{T} 55$ (in 123). Although by definition the complexity of any closed term is $O(1)$, the ability to determine automatically individual constants improves upon conventional manual analysis, where the tracking of constants is generally infeasible [5].

Given our definition of computational complexity, we cannot analyze a function simply by reducing its lambda term, as $\lambda x.b \downarrow \lambda x.b$ (in 0) for all $x,b$. Instead, we need to restate the informal notion of the complexity of a function $f$ more precisely as the cost of computing $f(x_1,\ldots,x_n)$ for all possible arguments $x_j$. To reflect this view, we supply $f$ with formal arguments $a_1,\ldots,a_n$ representing arbitrary type-correct terms and use the annotated semantics to reason about the cost of reducing $p$ for all $a_1,\ldots,a_n$. The complexity of the \texttt{fibo} function thus becomes the complexity of the term $\texttt{fibo}\ m$ for any $m$ of type $\mathbb{N}$. A simple proof by induction shows that $\texttt{fibo}\ m \downarrow \texttt{F}_m$ (in $3m + 4 + \texttt{F}_{m+1} + \hat{m}$), i.e., \texttt{fibo} is exponential in $m$.

**Proof.** We first show by induction on $m^*$ the stronger statement

$$
\text{ind}(m; \lambda x,y.x;i,z.\lambda x,y.z\ y\ (x+y))\ a\ b
\downarrow F_{m-1}\ a^* + F_m\ b^*
$$

(in $f(m,\hat{m},\hat{a},\hat{b})$),

where $f(m,m_c,a_c,b_c)=3m + 2 + F_{m+1} + m_c + F_{m-1}a_c + F_m b_c$. The base case $m^*=0$ follows immediately from

$$
\text{ind}(0; \lambda x,y.x;i,z.\lambda x,y.z\ y\ (x+y))\ a\ b
\downarrow a^*
$$

(in $3 + \hat{a}$),

where as usual $F_1=1$. For the step case, assume that

$$
\text{ind}((m-1)^*;\ldots)\ a\ b\downarrow F_{m-2}\ a^* + F_{m-1}\ b^*
$$

(in $f(m-1,0,\hat{a},\hat{b})$)

(note that in recursive calls, the principal argument is already reduced to canonical form). Then

$$
\text{ind}(m; \lambda x,y.x;i,z.\lambda x,y.z\ y\ (x+y))\ a\ b
\downarrow (\lambda x,y.\text{ind}((m-1)^*;\ldots)\ y\ (x+y))\ a\ b
\downarrow (m-1)^*;\ldots)\ b\ (a+b)
\downarrow F_{m-2}\ b^* + F_{m-1}(a+b)^*
$$

(in $f(m-1,0,\hat{b},\hat{a+b})$).

The value is equal to $F_{m-1}\ a^* + (F_{m-1}+F_{m-2})b^* = F_m\ a^* + F_m\ b^*$. The total complexity is

$$
1 + \hat{m} + 2 + f(m-1,0,\hat{b},\hat{a+b})
= 3 + \hat{m} + 3(m-1) + 2 + F_m + 0 + F_{m-2}\hat{b} + F_{m-1}\hat{a+b}
= 2 + \hat{m} + 3m + F_m + F_{m-2}\hat{b} + F_{m-1}(\hat{a} + \hat{b} + 1)
$$
\[\begin{align*}
&= 2 + \hat{m} + 3m + (F_m + F_{m-1}) + (F_{m-1} + F_{m-2})\hat{b} + F_{m-1} \hat{a} \\
&= f(m)
\end{align*}\]

The original claim now follows trivially from the definition of \(\text{fibo}\) and \(\hat{0} = \hat{1} = 0\). □

Terms like \(\text{fibo} \; m\) that involve meta variables are not proper terms of the language and are thus not understood by the evaluation system. Consequently, we cannot use an annotated semantics by itself to compute the complexity of meta terms automatically, even if we replace meta variables by regular term variables to obtain proper but open terms. We will show in Section 3 how we can extend annotated semantics to analyze meta terms automatically.

2.1. Expressing higher-order complexity

While we can express the computational complexity of first-order terms as generic as \(\text{add} (a; b)\) clearly and concisely by \(\hat{a} + \hat{b} + 1\), our development thus far falls short of higher-order terms. As an example, how should we describe the complexity of the second-order term \(\lambda g . \; g \; 0\) of type \((\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z}\)? Naive interpretation of \(\mathcal{F}\) would suggest the evaluation sequence

\[
(\lambda g . \; g \; 0) \downarrow f \; 0 \quad \text{(in 1)} \\
\downarrow f^* \; 0 \quad \text{(in } \hat{f}) \\
\downarrow\downarrow (f^*0) \quad \text{(in } \hat{f^*0})
\]

and thus that \(\lambda g . \; g \; 0\) \(\downarrow \hat{f}\). Although this is an intuitively meaningful description, an expression like \(\hat{f}^*0\) hardly tells us more than the original term \(\text{f} \; 0\). Obviously, unqualified use of the hat notation does not provide any insight into the computational behavior of a program, as \(\downarrow X^* \) (in \(\hat{X}\)) is vacuously true for even the most complicated computations.

In order to state the complexity of a higher-order meta term, we need some means to characterize the computational behavior of its instantiations that goes beyond the quotation of formal arguments. Ideally, any such calculus should be compositional, i.e., the complexity of any term should be a function of the complexity of its subterms. Our proposed solution to higher-order complexity called type decomposition breaks higher-order terms into the values and complexities of its constituent subtypes. The main idea is similar to Tait’s method of proving the normalizability of finite-type terms \(t\) by providing arguments \(a_1, \ldots, a_n\) such that \(t[a_1 \ldots a_n]\) is of atomic type [26]. Type decomposition is compositional with respect to value and complexity, i.e., the value and the complexity of a term are functions of the combined values and complexities of its subterms. Note that theories like \(\mathcal{N}\), in which integers are ground terms, cannot be compositional with respect to complexity alone, as the complexity of a recursive term \(\text{f} \; \text{ind} (m; b; i, z.s)\) necessarily depends on \(m^*\).

The decomposition \(\mathcal{D}(t)\) of a term \(t\) of base type \(\mathbb{Z}\) is simply its value \(t^*\) and its complexity \(\hat{t}\):

\[\mathcal{D}(t : \mathbb{Z}) = (t^*, \hat{t}).\]
Taken together, these two quantities sufficiently describe an arbitrary integer term in that the complexity of any closed first-order function \( g : \mathbb{Z} \rightarrow \mathbb{Z} \) can be expressed in terms of the decomposition \( \mathcal{D}(a) \) of its formal argument \( a \), i.e., \( \tilde{g} \tilde{a} = \phi(a^*, a) \) for some function \( \phi \).

For a first-order term \( f \) of type \( \mathbb{Z} \rightarrow \mathbb{Z} \), the full reduction of \( f \) to base type requires a formal argument \( a \) of type \( \mathbb{Z} \). If we assume that \( a \) has decomposition \( \mathcal{D}(a) = (a_v, a_c) \) for some unknown quantities \( a_v = a^*, a_c = \tilde{a} \), the decomposition of \( f \) becomes

\[
\mathcal{D}(\vdash f : \mathbb{Z} \rightarrow \mathbb{Z}) = (f^*, \hat{f}, \mathcal{D}(a) \mapsto f_1(\mathcal{D}(a)), \mathcal{D}(a) \mapsto f_0(\mathcal{D}(a))),
\]

for some functions \( f_1(a_v, a_c) = (f^*)^* \) and \( f_0(a_v, a_c) = \hat{f}^*a \) that are effectively computable from \( f \). Again, the complexity of any second-order function \( G \) can be expressed in terms of \( \mathcal{D}(f) \) as \( \mathcal{G} \hat{f} = \psi(f^*, \hat{f}, f_v, f_c) \) for some \( \psi \). Note that the simpler definition \( (f^*, \hat{f}, x \mapsto (f^*x)^*, x \mapsto \hat{f}^*x) \) is not compositional as it involves the actual term \( x \).

These examples easily generalize to higher-order functions. The decomposition of a term of type \( \tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_n \rightarrow \mathbb{Z} \) is a list of \( 2n + 2 \) functions

\[
(d_0 \mapsto f_{0c}(d_0), d_0 \mapsto f_{0c}(d_0), \ldots, d_n \mapsto f_{nc}(d_n), d_n \mapsto f_{nc}(d_n)),
\]

where each \( d_j \) is the decomposition \( \mathcal{D}(d_{j-1} \vdash a_j : \tau_j) \) of formal argument \( a_j \) and \( d_0 = \emptyset \) (cf. Fig. 2). The current ACA system also uses a simplified type decomposition \( \mathcal{D}' \) that omits references to values \( f^* \) that are not of base type.

While type decomposition is a function on terms, the structure of the decomposition depends uniformly on type. Given a formal argument \( a \) of type \( \tau \), we can assign an abstract type decomposition \( (a_0, \ldots, a_n) \) to \( a \) in order to reason meaningfully about the computational behavior of \( a \). As an example, consider the second-order function

\[
fibo' = \lambda n.\text{ind}(n; \lambda f.\text{if } 0 1; i.z.\lambda f.z (i x y f y (x + y)))
\]

of type \( \mathbb{N} \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z} \). The term generalizes the Fibonacci function in so far as \( \text{fibo}' m i x . y . x^1 \) computes \( F_m \). To analyze the function, we assume that \( f \) has simplified decomposition \( (f_0, x_c \mapsto f_1(x_c, y_c) \mapsto f_2) \), i.e.,

\[
f \downarrow f^* \quad \text{(in } f_0),
\]

\[
f^* u \downarrow (f^* u)^* \quad \text{(in } 1 + f_1(\tilde{u})), \text{ and}
\]

\[
(f^* u)^* v \downarrow ((f^* u)^* v)^* \quad \text{(in } 1 + f_2(\tilde{u}, \tilde{v})).
\]
for all terms \( u, v \) of type \( \mathbb{Z} \). Given this decomposition, the complexity of \( \text{fib} \) is 4\( m + 5 + \hat{m} + f_0 + f_1(F_{m+1} - 1) + f_2(F_{m+1} - 1, F_{m+2} - 1) \). Hence, the complexity of computing \( F_m \) using \( \text{fib} \) is 4\( m + 4 + \hat{m} + F_{m+1} \).

**Proof.** We first show by induction on \( m^* \) that

\[
\text{ind}(m; \lambda f.x 0 1; i, z, \lambda f, z (\lambda x, y. f (x+y))) \downarrow \downarrow \\
\text{in } 4m + 4 + \hat{m} + f_0 + f_1(F_{m+1} - 1) + f_2(F_{m+1} - 1, F_{m+2} - 1)
\]

for all \( m \). For the base case \( m^* = 0 \), we observe that

\[
\text{ind}(m; \lambda f.x 0 1; i, z, \lambda f, z (\lambda x, y. f (x+y))) \downarrow \\
\text{in } \hat{m} + 1 \\
\text{ind}(m; \lambda f.x 0 1; i, z, \lambda f, z (\lambda x, y. f (x+y))) \downarrow \\
\text{in } 2 + f_0 + f_1(\hat{0}) + f_2(\hat{0}, \hat{1})).
\]

For the step case \( m^* > 0 \), we similarly evaluate

\[
\text{ind}(m; \lambda f.x 0 1; i, z, \lambda f, z (\lambda x, y. f (x+y))) \downarrow \\
\text{in } m + 1 \\
\text{ind}(m; \lambda f.x 0 1; i, z, \lambda f, z (\lambda x, y. f (x+y))) \downarrow \\
\text{in } 2 + f_0 + f_1(\hat{0}) + f_2(\hat{0}, \hat{1})).
\]

To apply our inductive hypothesis, we need the decomposition of the argument term \( \lambda x, y. f y (x+y) \). Obviously, \( \lambda x, y. f y (x+y) \downarrow \downarrow \text{in } 0 \) and \( \lambda y. f y (x+y) \downarrow \downarrow \text{in } 0 \). Then

\[
f y (x+y) \\
f^* y (x+y) \text{ in } f_0 \\
f' (x+y) \text{ in } f_1(\hat{y}) + 1 \\
f'' \text{ in } f_2(\hat{y}, \hat{x} + \hat{y} + 1) + 1
\]

for some terms \( f', f'' \), yielding the simplified decomposition

\[
(f_0, f_1, f_2) = (0, x_c \mapsto 0, (x_c, y_c) \mapsto f_0 + f_1(y_c) + f_2(y_c, x_c + y_c + 1) + 2).
\]

Therefore, by our inductive hypothesis,

\[
\text{ind}((m-1)^*; \lambda f.x 0 1; i, z, \lambda f, z (\lambda x, y. f (x+y))) \downarrow \\
\text{in } 4(m-1) + 4 + (m-1)^* + 0 + f_0 + f_1(F_{m+1} - 1) + f_2(F_{m+1} - 1, F_{m+1} - 1 + 1) + 2).
\]

The original claim now follows trivially. \( \square \)

### 3. Symbolic evaluation

The previous section outlined our framework for manual complexity analysis of higher-order terms. Our approach to automation extends the computation of evaluation costs from closed terms to meta terms by generalizing a given annotated semantics \( \mathcal{A} \).
into a symbolical semantics $\mathcal{J}$ in which the meta variables of $\mathcal{A}$ become proper terms.

The complexity of a meta term $t$ with respect to $\mathcal{A}$ is then simply the computed cost of reducing $t$ to canonical form in $\mathcal{J}$.

The semantic interpretation of a symbolic term of $LDY^+ A$ is its set of proper term instantiations in $\mathcal{A}$, where we write $s_1 \equiv_{\mathcal{J}} s_2$ to denote that $s_1$ and $s_2$ have the same set of instantiations in $\mathcal{A}$. The symbolic extension of a given annotated semantics is open-ended in that we may add arbitrary terms and reduction rules to $LDY^+ A$ in order to capture the meta information generally present in informal complexity arguments. The central building block of any symbolic semantics is the symbolic term $\text{cpx}^*(m; t) \equiv \{ u \in A | u \downarrow A t' \text{ for some } m'; t' \equiv m; t \}$ for abstract computation

\[
\frac{t \downarrow w \text{ (in } n)}{\text{cpx}^*(m; t) \downarrow w \text{ (in } m + n)}.
\]

The $\text{cpx}^*$ term is mainly used to represent a class of terms that exhibit a certain computational behavior. The meta variables $v$ of type $\tau$ of $\mathcal{A}$ are encoded as parameterized symbolic terms $\text{var}^*(v: \tau) \equiv \{ t | t: \tau \}$, although we will continue to write $t[v]$ for brevity. The corresponding reduction rule for $\text{var}^*$ automates our idea of type decomposition.

Recall from the previous section that the simplified decomposition of a function $f$ of type $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ consists of three functions $f_0$, $x \mapsto f_1(x)$, and $(x, y) \mapsto f_2(x, y)$. If we replace meta variable $f$ by

$$\text{cpx}^*(f_0; \hat{x}. \text{cpx}^*(f_1(x)); \hat{y}. \text{cpx}^*(f_2(x, y); f_3(x, y)))),$$

symbolic evaluation will introduce the names of these functions into the complexity expression for $f$. There is exactly one such construct for each meta variable $v$ of type $\tau$, which we call the representative $R$ of $v$ (cf. Fig. 3):

\[
\frac{R(\vdash v: \tau) \downarrow w \text{ (in } n)}{v: \tau \downarrow w \text{ (in } n)} \quad (v \text{ meta variable}).
\]

Note that the variable indices generated by $R$ are sequences over the alphabet $\{i, o, 1, 2, v\}$, although we will continue to use numerical indices when convenient.
The symbolic semantics $\mathcal{F}$ cannot be generated mechanically from $\mathcal{A}$ in as much as $\mathcal{A}$ cannot be derived automatically from $\mathcal{F}$. The subsequent development chooses the symbolic extension of $\mathcal{F}$ to illustrate the design principles for symbolic semantics but is easily adopted to other annotated semantics as well. Our rules for $\mathcal{F}$ compute an upper bound on the complexity of terms with respect to $\mathcal{F}$, i.e., if $t \downarrow \mathcal{F} w$ (in $n$), then $t \downarrow \mathcal{F} w$ (in $m$) for some $m \leq n$ for all consistent instantiations. The interested reader can find a more comprehensive treatment of $\mathcal{F}$, including rules for conditionals and lazy lists, in [6,7]. We find it intriguing that although [7] is concerned with first-order terms only, most of the rules presented in that paper translate to higher-order terms without modification.

Symbolic arithmetic in $\mathcal{F}$ is straightforward. For each arithmetical operator $\mathsf{mul}$, $\mathsf{add}$, $\mathsf{sub}$, $\mathsf{div}$, $\mathsf{mod}$, a new symbolic term $\mathsf{mul}^*$, ... represents the canonical term denoting its integer value, i.e., $\mathsf{mul}^*(t_1; t_2) \equiv \mathsf{mul}(t_1; t_2)^*$. The corresponding symbolic reduction rules are

\[
\begin{align*}
\text{(arith)} & \quad u \downarrow k_1 \ (\text{in } n_1), \ v \downarrow k_2 \ (\text{in } n_2) \\
& \quad \mathsf{mul}(u; v) \downarrow \mathsf{mul}^*(k_1; k_2) \ (\text{in } n_1 + n_2 + 1) \quad (\text{also add, ...}).
\end{align*}
\]

The complexity of a primitive recursive term $\mathsf{ind}(m; b_2; i, z, s)^\tau$ of type $\tau$ is naturally expressed as a collection of recurrence equations over its principal argument $m$. To generate these equations, the symbolic evaluation rules identify the inductive term with the representative $\mathcal{R}(r(m) : \tau)$ for some fresh meta variable $r$. Reducing the representative in lieu of the inductive term,

\[
\begin{align*}
\text{(ind)} & \quad p \downarrow m \ (\text{in } n_1), \ \mathcal{R}(r(m) : \tau) \downarrow w \ (\text{in } n_2) \\
& \quad \mathsf{ind}(p; b_2; i, z, s) : \tau \downarrow w \ (\text{in } n_1 + n_2) \quad (r \text{ fresh variable}),
\end{align*}
\]

introduces the recurrence variables $r_0, r_1, \ldots$ as canonical names for the abstract type decomposition of $\tau$ into the complexity expression for $\mathsf{ind}$. Since the representative captures the computational behavior of the inductive term for all canonical $m$, we can replace the recursive call $\mathsf{ind}(m-1; b_2; i, z, s)^\tau$ by the non-recursive $\mathcal{R}(r(m-1) : \tau)$ when unfolding the step case $s[\ldots/z]$. To generate the recurrence equations describing the inductive term, we decompose base case $b$ and step case $s$

\[
\begin{align*}
\mathcal{D}(b : \tau) & = \{ b_0, x_0 \mapsto b_1, (x_0, x_1) \mapsto b_2, \ldots \}, \\
\mathcal{D}(s[m/i, \ \mathcal{R}(r(m-1) : \tau)/z] : \tau) & = \{ s_0, x_0 \mapsto s_1, (x_0, x_1) \mapsto s_2, \ldots \},
\end{align*}
\]

and match the resulting decomposition functions

\[
r_j(m, x_0, \ldots, x_{j-1}) = [j = 0] + \left\{ \begin{array}{ll} b_j(x_0, \ldots, x_{j-1}) & \text{if } m = 0, \\ s_j(x_0, \ldots, x_{j-1}) & \text{if } m > 0 \end{array} \right.
\]

to derive recursive expressions for $r_j(m)$. Note that the extra reduction step for $r_0$ accounts for the reduction of the $\mathsf{ind}$ term to $b$ or $s$.

All recurrence equations generated by decomposition are parameterized by higher-order functions $x_j$ that encode the computational behavior of the input arguments. Parameters are essential in describing the computational complexity of recursive higher-order functions, even if we need only a particular instance $x'_j$ of the closed solution
$r(m, x)$ for the final complexity expression. Alas, parameterized recurrence equations are considerably harder to solve than unparameterized ones. Thus, while our algorithm for generating higher-order recurrence equations from terms is complete, we will not be able to find closed solutions for all instances. This is, of course, to be expected, since complexity analysis is an undecidable problem.

**NUPRL** implements general recursion through recursive abstractions $ab(\bar{u}) \equiv \phi(ab, \bar{u})$, which are translated internally into \( Y (\lambda ab. \lambda \bar{u}. \phi(ab, \bar{u})) \bar{u} \). To infer the complexity of a term $t$ containing a recursive abstraction $ab$, we decompose the argument terms of $ab$

\[ (rec) \quad \frac{R(r(D(\bar{u})): \tau) \downarrow w \ (in \ n)} {ab(\bar{u}): \tau \downarrow w \ (in \ n)} \quad (r \ fresh \ variable) \]

and generate recurrence equations by decomposing the right-hand side of the recursive definition of $ab$:

\[ D(\phi(ab, \bar{u})) = (s_0, x_0 \mapsto s_1(x_0), \ldots). \]

Unlike in the primitive recursive case, however, we need to supply manually a collection of base cases $\bar{u}(1), \ldots, \bar{u}(n)$, from which the system generates additional equations

\[ D(\phi(ab, \bar{u}(k))) = (b_0(k), x_0 \mapsto b_1(k)(x_0), \ldots). \]

Since the base cases $\bar{u}(k)$ are more specific than $\bar{u}$, symbolic evaluation of $\phi(ab, \bar{u}(k))$ will generally reduce more conditionals non-symbolically and thus yield expressions $b_j(k)$ that are more specific than $s_j$. Combining $s_j$ and $b_j(k)$, we obtain multi-argument recurrence equations

\[ r_j(\bar{u}, x_0, \ldots, x_{j-1}) = \begin{cases} 
  b_j(1)(x_0, \ldots, x_{j-1}) & \text{if } \bar{u} = \bar{u}(1), \\
  b_j(2)(x_0, \ldots, x_{j-1}) & \text{if } \bar{u} = \bar{u}(2), \\
  \ldots & \ldots \\
  s_j(x_0, \ldots, x_{j-1}) & \text{otherwise}, 
\end{cases} \]

which are amenable to the multi-argument solver described in Section 4. Future versions of the ACA system might also include heuristics for guessing base cases $\bar{u}(k)$ automatically.

Evaluation of symbolic terms only reflects the complexity of meta terms if the symbolic semantics $\mathcal{S}$ are *sound* with respect to the underlying annotated semantics $\mathcal{A}$, i.e., if

\[ t \downarrow^\mathcal{S} w \ (in \ n) \Rightarrow t' \downarrow^\mathcal{A} w' \ (in \ m) \ & \ m \leq n' \]

for all consistent and type-correct instantiations $t', w', n'$ of symbolic terms $t, w, n$. It is important to realize, however, that the correctness of the ACA system does not rely on a soundness proof of $\mathcal{S}$, as the system automatically constructs a formal proof that independently asserts the correctness of the complexity bound in $\mathcal{S}$.
3.1. A symbolic evaluation example

We have seen in Section 2 that the complexity of $\text{fib}o^1_m$ is exponential in $m$. This might seem surprising given that $\text{fib}$ implements the “linear” algorithm (as opposed to the “exponential” version based on the recursive definition $F_m = F_{m-1} + F_{m-2}$). The source of this inefficiency is the call-by-name evaluation of internal arguments $x$ and $y$: in computing $F_m$, the algorithm builds a sum of exponentially many $\text{fib}o^1$ and $\text{fib}o^2$ terms whose subsequent reduction takes exponential time. To improve upon $\text{fib}$, we need to force evaluation of these intermediate sums by selective call-by-value evaluation:

\[
(\text{letval}) \quad t \downarrow u \ (\text{in} \ n_1) \quad \frac{b[u/x] \downarrow w \ (\text{in} \ n_2)}{\text{letval} \ x = t \ \text{in} \ b \downarrow w \ (\text{in} \ n_1 + n_2 + 1)}.
\]

While $\mathcal{N}$ does not support call-by-value evaluation of terms, we can simulate $\text{letval}$ at least for integer terms by using the following trick:

\[
\text{letval} \ x = t \ \text{in} \ b \downarrow \ \text{ind}(t; b[0]; i, z.b[i])
\]

We may then define a new function $\text{fib}o^1$ as

\[
\lambda n.\text{ind}(n; \lambda x, y, z. \text{letval} \ y' = y \ \text{in} \ z \ (y' + x)) \ \text{fib}o^1 m
\]

to obtain an efficient version of the Fibonacci function, as the following automated analysis will show.

The representative $R_m = \mathcal{D}(r(m) : Z \rightarrow Z \rightarrow Z)$ of above inductive term is

\[
\text{cpx}^*(r(m); \hat{r}_l.\text{cpx}^*(r_o(m, r_1); \hat{r}_o.\text{cpx}^*(r_{oo}(m, r_1, r_1); r_{oo})) \ 0 \ 1)
\]

so evaluation of $\text{fib}o^1 m$ using simplified decomposition yields

\[
\begin{align*}
\text{fib}o^1 m \\
\downarrow \text{ind}(m; \lambda x, y, z. \text{letval} \ y' = y \ \text{in} ...) \ 0 \ 1 (\text{in} \ m) \\
\downarrow \text{cpx}^*(r(m); \hat{r}_l.\text{cpx}^*(r_o(m, r_1); \hat{r}_o.\text{cpx}^*(r_{oo}(m, r_1, r_1); r_{oo})) \ 0 \ 1 (\text{in} \ m) \\
\downarrow \text{cpx}^*(r_o(m, \mathcal{D}(r_0^0)) \ (\text{in} \ r(m) + 1) \\
\downarrow \text{cpx}^*(r_{oo}(m, 0, \mathcal{D}(r_1^0)); r_{oo}) \ (\text{in} \ r_o(m, 0) + 1) \\
\downarrow r_{oo} \ (\text{in} \ r_{oo}(m, 0, 0))
\end{align*}
\]

with a preliminary total complexity of $r(m) + r_o(m, 0) + r_{oo}(m, 0, 0) + 4 + m$. Recurrence equations for $r$, $r_o$, and $r_{oo}$ are generated from the simplified decompositions of base and step case

\[
\begin{align*}
\mathcal{D}(\lambda x, y, z. x) &= (0, x_c \mapsto 0, (x_c, y_c) \mapsto x_c) \\
\mathcal{D}(\lambda x, y. \text{letval} \ y' = y \ \text{in} \ R_{m-1} \ y' (x + y')) &= (0, x_c \mapsto 0, (x_c, y_c) \mapsto r(m - 1) + x_c + r_o(m - 1, 0) + r_{oo}(m - 1, 0, x_c + 1) + 3),
\end{align*}
\]
where the latter derives from the evaluation sequence

\[
\begin{align*}
\text{letval } y' = y & \text{ in } R_{m-1} y' (x+y') \\
\downarrow R_{m-1} y^* (x+y^*) & \quad \text{(in } \hat{y} + 1) \\
= \text{cpx}^*(r(m-1); \hat{c}r_1, \text{cpx}^*(r_o(m-1, r_1)); \hat{c}r_{oo} \cdot \text{cpx}^*(r_{oo}(m-1, r_1; r_{oo})); r_{oo})) y^* (x+y^*) \\
\downarrow \text{cpx}^*(r_o(m-1, \mathcal{D}'(r y^*))) & \quad \text{(in } r(m-1) + 1) \\
\downarrow \text{cpx}^*(r_{oo}(m-1, 0, \mathcal{D}'(r x+y^*)); r_{oo}) & \quad \text{(in } r_o(m-1, 0) + 1) \\
\downarrow \downarrow r_{oo} & \quad \text{(in } r_{oo}(m-1, 0, \hat{x} + 1)) \\
\end{align*}
\]

Combined, this yields recurrence equations

\[
\begin{align*}
\quad r(m) = \begin{cases} 
1 & \text{if } m = 0, \\
1 & \text{if } m > 0,
\end{cases} \\
\quad r_o(m, x_c) = \begin{cases} 
0 & \text{if } m = 0, \\
0 & \text{if } m > 0,
\end{cases} \\
\quad r_{oo}(m, x_c, y_c) = \begin{cases} 
x_c & \text{if } m = 0, \\
3 + y_c + r(m-1) + r_o(m-1, 0) + r_{oo}(m-1, 0, x_c + 1) & \text{if } m > 0,
\end{cases}
\end{align*}
\]

where \( r \) includes the adjustment for the \text{ind term reduction. Obviously, } r \equiv 1 \text{ and } r_o \equiv 0, \text{ so the last equation simplifies to}

\[
\begin{align*}
\quad r_{oo}(m, x_c, y_c) = \begin{cases} 
x_c & \text{if } m = 0, \\
4 + y_c + r_{oo}(m-1, 0, x_c + 1) & \text{if } m > 0.
\end{cases}
\end{align*}
\]

The reader can easily verify that the expression

\[
5 * m - 1 + x + y - \text{if } m=0 \text{ then } y - 1 \text{ else } 0 \text{ fi} \\
- \text{if } m=1 \text{ then } x \text{ else } 0 \text{ fi}
\]

returned by the ACA recurrence solver is a closed solution for \( r_{oo}(m, x, y) \). Consequently, the complexity of 'fibo\(^v m\)' is \( 5m + 4 + [m = 0] + m \).

4. Higher-order recurrence equations

As we saw in the previous section, the symbolic evaluation of inductive terms yields systems of multivariate recurrence equations (REs) whose general form is given by

\[
R(m, f_1, \ldots, f_n) = \begin{cases} 
F(f_1, \ldots, f_n) & \text{if } m = 0, \\
G(f_1, \ldots, f_n, m, R) & \text{if } m > 0.
\end{cases}
\]

We call \( m \) the argument and \( f_j \) the parameters of \( R \). Parameters are higher-order functions whose types \( \tau_j \) are elements of the recursively defined family \( T = T \times \cdots \times T \rightarrow R \lor R \). If all parameters are scalars of type \( R \), we say that
$R$ is a first-order equation. A simple example for a second-order RE is

$$R(m, f) = \begin{cases} f(0) & \text{if } m = 0, \\ R(m-1, x) \mapsto f(2x + 1) + x + 1 & \text{if } m > 0, \end{cases}$$

for which $R(m, f) = f(2^m - 1) + 2^m - 1$ is a closed solution. Note that our terminology differs from conventional usage, where the order usually refers to the argument span $a$ of the recursive calls $R(m-1, \bar{x}), \ldots, R(m-a, \bar{x})$ in $G$.

Our main idea for solving higher-order REs is to reduce a higher-order equation to a first-order equation by converting the parameter functions $f_j$ into polynomials $p_j = \mathcal{P}(f_j)$ that can be represented by their coefficient lists $\mathcal{C}(p_j)$. More precisely, we establish one-to-one mappings $\mathcal{P}$ and $\mathcal{C}$ between certain admissible higher-order functions, polynomials, and lists of real numbers

$$\mathcal{C}(g) : \mathcal{P}(f) \mapsto \mathcal{C}(p) : \mathcal{P}^m \rightarrow \mathbb{R} \mapsto (c_1, \ldots, c_n) : \mathbb{R}^n$$

such that $f(g) = \mathcal{P}(f)(\mathcal{C}(p))$ for all admissible functions $f : \bar{x} \rightarrow \mathbb{R}$ and arguments $g : \bar{x}$. We call $\mathcal{P}(f)$ the polynomialization of $f$. If necessary, we write $\mathcal{P}_d(f)$ to indicate the degree $d$ of all polynomials involved in the polynomialization of $f$.

The polynomialization of a scalar $f : \mathbb{R} \rightarrow \mathbb{R}$ is itself for any degree $d$. For a function $f : \bar{x} \rightarrow \mathbb{R}$, we recursively polynomialize the formal arguments $\bar{a} : \bar{x}$ to derive an assumed coefficient list $\mathcal{C}(\mathcal{P}(a_1 : \tau_1)) \oplus \cdots \oplus \mathcal{C}(\mathcal{P}(a_n : \tau_n)) = (c_0, \ldots, c_n)$, where $n$ depends on $\bar{x}$ and $d$. The polynomialization of $f$ is then a polynomial over $\mathcal{C}$

$$\mathcal{P}_d(f) = (c_0, c_1, \ldots, c_n) \mapsto C_{000..00} + C_{100..00}c_0 + C_{010..00}c_1 + \cdots + C_{000..01}c_n + C_{200..00}c_0^2 + C_{110..00}c_0c_1 + \cdots + C_{000..02}c_n^2 + \cdots + C_{d00..00}c_0^d + C_{(d-1)10..00}c_0^{d-1}c_1 + \cdots + C_{000..0d}c_n^d,$$

which is completely characterized by its $\sum_{i=0}^{d} \binom{n+i}{d}$ coefficients $C_j$.

Returning to our second-order example from above, the polynomialization of $x \mapsto f(2x + 1) + x$ for an assumed linear function $f(x) = f_0x + f_0$ is $x \mapsto 2f_1x + f_0 + f_0 + x$. The polynomialized REs becomes

$$R(m, f_1, f_0) = \begin{cases} f_0 & \text{if } m = 0, \\ R(m-1, 2f_1 + 1, f_0 + f_1) + 1 & \text{if } m > 0, \end{cases}$$

which is a first-order equation.

It is important to note that the polynomialization of a given parameter $f_j$ need not exist. Remember from the previous section, however, that the parameters encode the computational complexity of formal arguments $a_j$ passed into the inductive term $t$ characterized by $R$. Thus, the complexity $f_j$ is introduced into $R$ if and only if $t$ evaluates its argument $a_j$. In most cases, the overall complexity $R$ will be a linear
function of $f_j$ that depends on how many times $t$ evaluates $a_j$. For fixed nested function calls, i.e., arguments $a_i, a_j$ passed into each other, this function might be of higher but still constant degree. Only if the nesting level depends on some value $a_j^*$ does the polynomialization not exist.

4.1. Solving parameterized first-order recurrences

Polynomialization reduces higher-order REs to parameterized first-order equations $R(m, \tilde{x})$ in which all parameters $x_j$ are scalars. Alas, there is no complete algorithm for obtaining closed solutions to first-order REs, even in the unparameterized case. Computer algebra systems such as MATHEMATICA cannot escape this fundamental limitation, but they incorporate the latest expert knowledge to solve the most important classes of frequently arising unparameterized equations. To harness that knowledge, we rewrite parameterized equations into systems of unparameterized first-order REs that can be solved by conventional means.

The principal idea—and the main limitation—of our method is the presumption of a particular form for the closed solution that is parameterized by functions of the argument $m$. In our current implementation, this assumed solution is a polynomial in $\tilde{x}$ whose coefficients are functions of $m$. It is important to realize, however, that because of these parameterized coefficients the polynomial $R(m, \tilde{x})$ may be an arbitrary function of $m$; in particular, we are not restricted to polynomial solutions. The recurrence solver of the ACA system first infers heuristically an upper bound $d$ on the degree of the solution and then sets up the general solution

$$R(m, x_1, \ldots, x_n) = \sum c_{i_1 \ldots i_n}(m)x_1^{i_1}x_2^{i_2} \cdots x_n^{i_n} \quad \text{such that } \left( \sum_j i_j \leq d \right),$$

which is applied to both sides of the RE. Matching the coefficients of each term $x_1^{i_1}x_2^{i_2} \cdots x_n^{i_n}$ yields a system of $\binom{n+d}{n}$ unparameterized recurrence equations, which can be solved by MATHEMATICA.

Concluding our second-order example from before, we assume that $R$ has closed solution $R(m, f_1, f_0) = c_{10}(m)f_1 + c_{01}(m)f_0 + c_{00}(m)$. Substituting this assumption into the polynomialized RE, we arrive at

$$c_{10}(m)f_1 + c_{01}(m)f_0 + c_{00}(m) = c_{10}(m-1)(2f_1 + 1) + c_{01}(m-1)(f_0 + f_1) + c_{00}(m-1) + 1,$$

$$c_{10}(0)f_1 + c_{01}(0)f_0 + c_{00}(0) = f_0.$$

Comparing the coefficients of the polynomial over $f_0$ and $f_1$ yields a system of recurrence equations

$$c_{10}(m) = 2c_{10}(m-1) + c_{01}(m-1), \quad c_{10}(0) = 0,$$
$$c_{01}(m) = c_{01}(m-1), \quad c_{01}(0) = 1,$$
$$c_{00}(m) = c_{10}(m-1) + c_{00}(m-1) + 1, \quad c_{00}(0) = 0.$$
which is solved by Mathematica to give us
\[ c_{10}(m) = c_{00}(m) = 2^m - 1, \quad c_{01}(m) = 1 \]
and thus 
\[ R(m, f_1, f_0) = (2^m - 1)f_1 + f_0 + 2^m - 1 \]
as the closed solution for our second-order example.

The first-order recurrence solver is easily generalized to solve multi-argument equations generated by general recursion. The only requirement is that the solution be polynomial in all but one argument, which has to be chosen as the argument for coefficients \( c_j \).

5. Examples

The pigeonhole principle, usually attributed to Dirichlet who used the term Schubfachprinzip in 1834, asserts that if \( m \) objects are placed into \( n \) containers, then there exist at least two objects that share the same container. For the special case \( m = n + 1 \), this is formally stated as
\[
\forall n \in \mathbb{N}^+ . \ \forall f : \mathbb{N}_{n+1} \to \mathbb{N}_n . \ \exists x : \mathbb{N}_{n+1} . \ \exists y : \mathbb{N}_n . \ f(x) = f(y).
\]

An algorithm extracted from a proof of this theorem takes a positive integer \( n \) and a function \( f : \mathbb{N}_{n+1} \to \mathbb{N}_n \) and returns a pair \( (x_1, x_2) \) of distinct integers such that \( f(x_1) = f(x_2) \). In this section we reproduce some sample runs of the ACA system analyzing three different implementations of the pigeonhole principle. Due to space limitations, the corresponding formal proofs had to be omitted from this paper but can be found in [6] and the standard ACA system distribution available from the Nuprl web site [27].

5.1. The inefficient proof

In program synthesis as well as in conventional program development, mathematical efficiency often comes at the expense of computational efficiency, and vice versa. The original proof of the pigeonhole principle that appeared in [14] is of great mathematical beauty but results in an infeasible algorithm.

The proof proceeds by induction on \( n \). The base case \( n = 1 \) is trivial, since necessarily \( f(0) = f(1) = 0 \). For the step case \( n > 1 \), the function \( f \) ranges over \( \{0, \ldots, n - 1\} \). We first search for some \( k < n \) such that \( f(n) = f(k) \) holds. If this search succeeds, we are done. If no such \( k \) exists, however, we define
\[
\begin{align*}
f'(x) &:= \begin{cases} f(n) & \text{if } f(x) = n - 1, \\ f(x) & \text{otherwise.} \end{cases}
\end{align*}
\]
This new function \( f'(x) \) ranges over \( \mathbb{N}_{n-1} \) for \( x = 0, \ldots, n - 1 \), for if \( f'(x') = n - 1 \) for some \( x' \), then \( f'(x') = f(n) = n - 1 \) and thus \( f(k) \neq n - 1 \) for all \( k < n \) by our previous check. Hence, we can apply our inductive hypothesis to obtain a pair \( x', y' \) such that \( f'(x') = f'(y') \), from which we can easily construct \( x, y \) for which \( f(x) = f(y) \) holds.
A ph_orig ==
\lambda n.\text{Ind}(\langle n - 1 \rangle + 1)
\begin{align*}
\text{base: } & \lambda n.s.n.1.Ax \\
\text{step: } & \text{zzj},n.2. \lambda n.s.n.200,n.3.
\end{align*}
\text{case if } n=1 \text{ then } \text{inl } Ax \text{ else inr } (\lambda.Ax) 
\begin{align*}
of \text{inl}(\langle 4 \rangle) & \Rightarrow \lambda f.\langle 0,1,Ax \rangle \\
| \text{inr}(\langle 5 \rangle) & \Rightarrow \lambda f.\text{case}
\end{align*}
\begin{align*}
\text{case if } n<0 \text{ then } \text{inr } (\lambda.Ax) \text{ else inl } (\lambda.Ax) 
\begin{align*}
of \text{inl}(\langle 2 \rangle) & \Rightarrow \text{Ind}(\langle n - 0 \rangle + 1) \\
\text{base: } & \lambda n.s.n.1.Ax \\
\text{step: } & \text{zzj},n.2. \lambda n.s.n.200,n.3.
\end{align*}
\text{case if } n=0 \text{ then } \text{inl } Ax \text{ else inr } (\lambda.Ax) 
\begin{align*}
of \text{inl}(\langle 4 \rangle) & \Rightarrow \lambda n.\text{inr } (\lambda.Ax) \\
| \text{inr}(\langle 5 \rangle) & \Rightarrow \text{case}
\end{align*}
\text{case if } (f n)=(f n@0) \text{ then } \text{inl } Ax \text{ else inr } (\lambda.Ax) 
\begin{align*}
of \text{inl}(\langle 10 \rangle) & \Rightarrow \text{inl } \langle n00,10 \rangle \\
| \text{inr}(\langle 11 \rangle) & \Rightarrow \text{inr } (\lambda.Ax) \\
of \text{inl}(\langle 15 \rangle) & \Rightarrow Ax \\
| \text{inr}(\langle 16 \rangle) & \Rightarrow (\lambda.Ax)\langle n - 0 \rangle Ax 0 Ax (\lambda.Ax)
\end{align*}
of \text{inl}(\langle 2 \rangle) \Rightarrow \text{let } \langle i,1,Ax \rangle = (\lambda (\lambda.Ax) (n - 1) Ax (n - 1) Ax \\
in \text{let } \langle j,9 \rangle = \%8 in \langle i, j, Ax \rangle \\
(\lambda.Ax)\langle n - 1 \rangle Ax n Ax
\end{align*}

Fig. 4. Extract of inefficient pigeonhole proof.

The ACA analysis of the extract shown in Fig. 4 applied to formal arguments \( m: \mathbb{N} \) and \( f: \mathbb{N} \rightarrow \mathbb{N} \) yields
\begin{align*}
M> & \text{cc } '\text{ph_orig m f}' ;; \\
1/2 & \ast (-94 + -2 \ast m + 15 \ast 2^{-}(3 + m) \\
& + 4 \ast f._{-} \ast (-3 + 3 \ast 2^{-} m + -1 \ast m) \\
& + 4 \ast f.o0 \ast (-3 + 3 \ast 2^{-} m + -1 \ast m) \\
& + f.o1 \ast (-32 + -9 \ast m + -1 \ast m^{-2} \\
& + 2^{-}(5 + m) + -13 \ast 2^{-} m \ast m + 3 \ast 2^{-} m \ast m^{-2})
\end{align*}
\( R^1_i(p_{ao}, p_1, p_0, p_1, p_0, m) = \)
\[
26 + 4 p_1^i + R^2_i(0, 0, p_1^i + 1) \\
+ R^1_i(1, 2 p_1^i, 2(p_{ao}^i + p_1^i + 4) + p_1^i(p_1^i + 1), p_1^i + 1, p_1^i - 1, m - 1)
\]
\( R^2_i(p_{ao}, p_1, p_0, p_1, p_0, m) = \)
\[
\max(R^2_i(0, 0, p_1^i + 1), \\
R^1_i(1, 2 p_1^i, 2(p_{ao}^i + p_1^i + 4) + p_1^i(p_1^i + 1), p_1^i + 1, p_1^i - 1, m - 1))
\]
\( R^1_i(p_{ao}, p_1, p_0, p_1, p_0, p_2, m) = \)
\[
\max(p_1^i, R^1_i(1, 2 p_1^i, 2(p_{ao}^i + p_1^i + 4) + p_1^i(p_1^i + 1), p_1^i + 1, p_1^i - 1, m - 1))
\]
\( R^2_i(p_1^2, p_2^2, m) = \)
\[
16 + p_1^i + 2 p_{ao}^i + 2 p_0^i + p_1^i p_1^i + p_2^i + p_1^i p_2^i + R^2_i(p_1^i + 1, p_2^i + 1, m - 1)
\]
\( R^2_i(p_1^2, p_2^2, m) = \)
\[
\max(p_1^i, R^2_i(p_1^i + 1, p_2^i + 1, m - 1))
\]
\( R^2_i(p_1^2 + 1, p_2^i + 1, m - 1)
\]
\( R^k_i(\ldots, 0) = [j = 0]\)

Fig. 5. Non-trivial recurrence equations for inefficient extract.

\[
+ m_\times ( \neg 8 + m + -1 + m \times 2 + 3 \times 2^\neg (2 + m) \\
+ f_{\times o1} \times m_\times (10 + 2 \times m + -5 \times 2^\neg (1 + m) \\
+ 3 \times 2^\neg (1 + m) \times m))
\]

where \(m_\times\) denotes \(m\) and \((f_{\times o1} \times f_{\times o1} + f_{\times o0})\) is the polynomialized decomposition of \(f\). We see that the algorithm is exponential in \(m\) even for functions \(f\) that do not evaluate their arguments (i.e., \(f_{\times o1} = 0\)).

In the course of the analysis, the system generates a total of 23 recurrence equations, of which Fig. 5 shows the non-trivial ones. The max terms are introduced by symbolic evaluation of conditionals (cf. [6] for details). Obtaining the closed solution shown above takes about 3 minutes on an Intel Pentium III machine and involves about 2500 calls to Mathematica.

5.2. The optimized proof

While the inefficient proof is concise and elegant, each evaluation of the function \(f'(x)\) requires two evaluations of the parent function \(f\). Thus, if we use call-by-name reduction for computing \(f'\), the evaluation tree grows exponentially in \(n\). An improved version of the proof, formalized in [21], eliminates the need for function \(f'\) by strengthening the inductive hypothesis to

\[
\forall k: \mathbb{N}_{n + 1}. \left(\forall l: \{k + 1, \ldots, n\}. \forall f': \mathbb{N}_{n'}. f_i' \neq f_j' \right) \\
\Rightarrow (\exists i: \mathbb{N}_{k + 1}. \exists j: \mathbb{N}_i. f_i = f_j).
\]

The resulting extract (cf. Fig 6) contains a loop \(k = n, n - 1, \ldots, 1\) that in each iteration checks for some \(x \in \{0, \ldots, k - 1\}, y \in \{k, \ldots, n\}\) for which \(f(x) = f(y)\) holds. While
Fig. 6. Extract of optimized pigeonhole proof.
this solution is rather pedestrian from a mathematical point of view, it is computationally very efficient, as the system’s analysis shows:

\[
M> \text{cc } \text{ph\_opt } m \ f^l \ ;;
\]
\[
1/6 * (408 + 322 * m + 66 * m^2 + 2 * m^3 + 3 * m * (18 * + 11 * m + m^2)) + 6 * f_0*(2 + 3 * m + m^2)
\]
\[
+ 6 * f_00*(2 + 3 * m + m^2)
\]
\[
+ 2 * f_01*(2 * m + 3 * m^2 + m^3)
\]
\[
+ 3 * f_01 * m_*(2 + 3 * m + m^2))
\]

: term

To obtain this solution, the system creates 42 recurrence equations, of which 12 are non-trivial, and makes about 5900 calls to MATHMATICA.

When we first analyzed the extract of the optimized proof, we were surprised to learn that the system still classified it as exponential. As it turned out, the optimized proof reused part of the inefficient proof to discharge some subgoals with contradicting assumptions, thus introducing an inefficient code fragment into the extract. By the very construction of the proof, this subroutine was unreachable for any type-correct input and thus did not incur a performance penalty. As it happened, passing an invalid function to the pigeonhole algorithm did execute the offending subroutine—hardly a concern, since the result would be meaningless anyway. Unfortunately, the ACA system currently does not support subtypes of \( \mathbb{N} \) and thus typed the input argument \( f \) as \( \mathbb{N} \rightarrow \mathbb{N} \). In this generality, however, the code was no longer dead and had to be included in the analysis. Once this unexpected behavior was discovered, a simple revision of the computationally expensive subproof yielded above result.

5.3. An alternative proof

In the aftermath of the pigeonhole incident, Allen independently developed an alternative proof [27] that was specifically designed to yield the obvious two-loop algorithm. Since it is derived from the slightly different specification

\[
\forall m: \mathbb{N}. \forall k: \mathbb{N}. \forall f: \mathbb{N} \rightarrow \mathbb{N}. \exists x, y: \mathbb{N}. x \neq y \land f(x) = f(y)
\]

the extract (cf. Fig 7) expects a dummy proof term \( \text{Ax}^1 \) for \( k < m \), which does not have any computational content and is ignored. The analysis returns

\[
M> \text{cc } \text{ph\_alt } (m + 1) \ m \ f \ Ax^1 \ ;;
\]
\[
23 + 3 * m_+ + 1/2 * (2 + m) *
\]
\[
(62 + 19 * (2 + m))
\]
\[
+ 2 * m_+
\]
\[
+ 2 * f_0 * (3 + m)
\]
\[
+ 2 * f_00 * (3 + m)
\]
\[
+ -1 * f_01
\]
\[
+ (1 + f_01) * (2 + m)^2)
\]

: term
A ph_alt ==
λm,k,f,.%
let <x,%5> = case case if m<0 then inr (λ,.Ax) else inl (λ.Ax)
  of inl(%2) ⇒ Ind((m - 0) + 1)
    base: λ%2,s,%1.Ax
  step: zzj,%2. λ%4,s,%2@0,n,%3.
    case if n=m then inl Ax else inr (λ.Ax)
    of inl(%4) ⇒ λ%3.inr (λ%4.let <k,%5> = %4 in Ax)
      | inr(%5) ⇒ λ%4@0.
    case %2 (λ.Ax) (m - (n + 1)) Ax (n + 1) Ax (λ.Ax)
    of inl(%6) ⇒ let <k,%7> = %6 in inl <k, %7>
      | inr(%7) ⇒ case
        case if n<0 then inr (λ,.Ax) else inl (λ.Ax)
        of inl(%2) ⇒ Ind((n - 0) + 1)
          base: λ%2,s,%1.Ax
        step: zzj,%2. λ%4,s,%2@0,n@0,%3.
          case if n@0=n then inl Ax else inr (λ.Ax)
          of inl(%4) ⇒ λ%3.inr (λ%4.let <k,%5> = %4 in Ax)
            | inr(%5) ⇒ λ%4@0.
          case %2 (λ.Ax) (n - (n@0 + 1)) Ax
            (n@0 + 1) Ax (λ.Ax)
          of inl(%6) ⇒ let <k,%7> = %6 in inl <k, %7>
            | inr(%7) ⇒ case if (f n)=(f n@0)
              then inl Ax else inr (λ.Ax)
              of inl(%10) ⇒ inl <n0, %10>
                | inr(%11) ⇒ inr (λ%12.let <k,%13> = %12 in
                  case if k=n@0 then inl Ax else inr (λ.Ax)
                  of inl(%15) ⇒ Ax
                    | inr(%16) ⇒ any (%7 <k, %13>)
                      (λ.Ax) (n - 0) Ax 0 Ax (λ.Ax)
                | inr(%3) ⇒ inr (λ%4.let <k,%5> = %4 in Ax)
                  of inl(%10) ⇒ inl <n, %10>
                    | inr(%11) ⇒ inr (λ%12.let <k,%13> = %12 in
                  case if k=n then inl Ax else inr (λ.Ax)
                  of inl(%15) ⇒ Ax
                    | inr(%16) ⇒ any (%7 <k, %13>)
                      (λ.Ax) (m - 0) Ax 0 Ax (λ.Ax)
                | inr(%3) ⇒ inr (λ%4.let <k,%5> = %4 in Ax)
                  of inl(%2) ⇒ %2
                    | inr(%3) ⇒ any any Ax
                  in <x, let <y,%6> = %5 in <y, λ.Ax, Ax>>

Fig. 7. Extract of alternate pigeonhole proof.
The system creates 26 recurrence equations, of which 8 are non-trivial, and makes about 2000 calls to MathematICA.

6. Conclusion

We introduced a general framework for reasoning about the computational complexity of higher-order functional programs. Cost models are based on operational semantics annotated with complexity information and measure the complexity of reducing meta terms to canonical form. For higher-order terms, we use type decomposition to express the complexity of formal input arguments. As an example, we provided $T$ for measuring the exact time complexity of functional programs with respect to NUPRL’s standard call-by-name evaluator.

We automated the analysis within our calculus by extending annotated semantics to meta terms. The symbolic evaluation of recursive terms generates systems of parameterized higher-order recurrence equations, which are decomposed into first-order equations by polynomialization. By assuming a certain generic solution, we can rely on the MathematICA computer algebra system to obtain closed solutions for these equations. Our reference implementation for the symbolic extension of $T$ is capable of analyzing non-trivial programs such as the pigeonhole algorithm.

We are currently completing the ACA system to generate automatically formal NUPRL proofs that assert the correctness of the complexity result inferred by symbolic evaluation. Future work might include the formalization and annotation of alternative evaluation strategies, in particular call-by-need, and the improvement of the recurrence solver by assuming more general solutions.

Acknowledgements

The author is much indebted to his adviser Robert L. Constable for his constructive feedback during this work. I also would like to thank Aleksey Nogin and Stuart Allen for providing versions of their pigeonhole proofs. Finally, I would like to thank the referees for their detailed and insightful suggestions on how to improve this paper.

References

[29] B. Wegbreit, Mechanical program analysis, Comm. ACM 18 (9).