Automated Complexity Analysis of Nuprl Extracted Programs

Ralph Benzinger

Department of Computer Science, Cornell University, Ithaca, NY 14853
(*e-mail:* ralph@cs.cornell.edu)

Abstract

This paper describes the *Automated Complexity Analysis Prototype* (ACA*p*) system for automated complexity analysis of functional programs synthesized with the NUPRL proof development system.

We define a simple abstract cost model for NUPRL's term language based on the current call-by-name evaluator. The framework uses *abstract functions* and *abstract lists* to facilitate reasoning about primitive recursive programs with first-order functions, lazy lists, and a subclass of higher-order functions.

The ACA*p* system automatically derives upper bounds on the time complexity of NUPRL extracts relative to a given profiling semantics. Analysis proceeds by abstract interpretation of the extract, where *symbolic evaluation rules* extend standard evaluation to terms with free variables. Symbolic evaluation of recursive programs generates systems of multivariable difference equations, which are solved using the MATHEMATICA computer algebra system.

The use of the system is exemplified by analyzing a proof extract that computes the maximum segment sum of a list and a functional program that determines the minimum of a list via sorting. For both results, we compare call-by-name to call-by-value evaluation.

1 Introduction

Automated tools for program analysis (Owre *et al.*, 1992; Gordon & Melham, 1993; Paulson, 1994) and synthesis (Coquand & Huet, 1985; Constable *et al.* , 1986; Smith, 1990) aid the development of provably correct programs. Over the past decade, many of these systems have matured to a stage where they become increasingly attractive for commercial, large-scale software projects. But despite these advances in program correctness, little attention has been paid to the quality of the generated programs in terms of resource efficiency. Formal specification methods (Hoare, 1969; Bjørner & Jones, 1978; Spivey, 1992) lack the machinery to make explicit statements about running time or space requirements during program execution. An algorithm generated by an interactive program synthesizer like NUPRL is guaranteed to conform to its specification, but controlling the resource consumption of the resulting code requires great skill and experience from the user performing the synthesis.

Complexity analysis of functional programs can be a hard and error-prone task. In (Constable *et al.*, 1998), Constable et al. formalized a large portion of finite

automata theory and synthesized a provably correct algorithm for state minimization. It was only later discovered that due to an inefficient proof of the Pigeon Hole Principle, the running time of the minimization algorithm was hyperexponential in the number of states (Nogin, 1997).

Machine-generated programs are particularly hard to penetrate without computer assistance. Accordingly, automated complexity analysis is a perennial yet surprisingly disregarded aspect of static program analysis. The seminal contribution to this area was Wegbreit's METRIC system (Wegbreit, 1975), which even today still represents the state-of-the-art in many aspects. METRIC can analyze simple programs written in a first-order subset of LISP and returns an upper bound on the worst-case complexity, a lower bound on the best-case complexity, and the expected complexity together with its variance. It assumes that all tests in the program are independent, and requires the user to provide probability distributions for each conditional statement.

The ACE system (Métayer, 1988) for analyzing FP programs uses a library of more than 1,000 transformation rules to translate a given program P into a *step-counting version* P' that for each argument $\bar{x}$ returns the number of primitive reduction steps required for computing $P(\bar{x})$. The program P' is then further transformed into a composition of known complexity functions.

Rosendahl (Rosendahl, 1986; Rosendahl, 1989) 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. The system can theoretically analyze programs written in other languages as long as the step-counting version can be formulated in first-order LISP.

Theoretical advances for analyzing lazy functional languages were made by Wadler (Wadler, 1989) and Bjerner and Holmström (Bjerner & Holmström, 1989). In essence, the authors use *projections* and *demand analysis*, a specifically tailored strictness analysis, to model an informal call-by-need reduction strategy for the untyped lambda calculus. Sands (Sands, 1990; Sands, 1995) introduces *cost closures* to reason about higher-order functions.

Other work includes the performance compiler of (Hickey & Cohen, 1988) for imperative and FP programs, and the ambitious $\Lambda\Upsilon\Omega$ system (Flajolet *et al.*, 1990) for average-case complexity.

1.1 The Nuprl *System*

The NUPRL proof development system (Constable *et al.* , 1986) is an automated theorem prover based on an extension of Martin-Löf type theory (Martin-Löf, 1979), a predicative type theory encoding constructive logic via the *propositions-as-types* principle: A proposition P is a type $[\![P]\!]$ whose members are the proofs of P. The *proof-as-programs* principle associates a proof pf of a theorem $\vdash P$ with a program p of type $[\![P]\!]$. NUPRL's underlying term language is an implicitly typed variant of the lambda calculus. Programs synthesized from proofs are closed expressions called

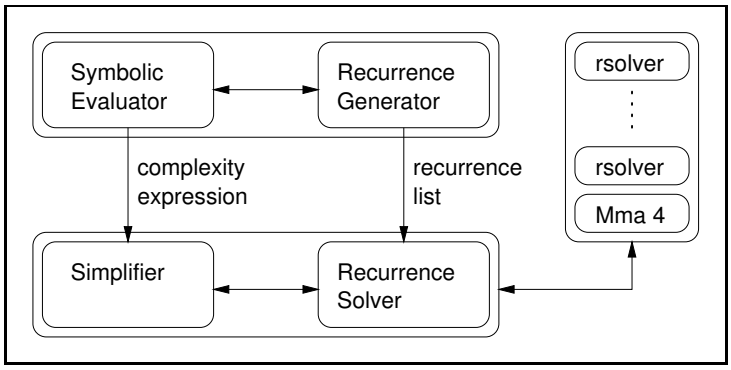

Fig. 1. The ACA*p* system

proof extracts that can be executed by NUPRL's evaluator, which uses a left-most outermost call-by-name reduction strategy to evaluate terms. The successor system METAPRL (Hickey, 2000) allows for different reduction strategies and includes an improved evaluator that uses explicit substitution and sharing for highly efficient call-by-need reduction.

Extracts are usually obfuscated by large portions of *dead code* generated by arithmetical reasoning and existential quantification. An element of $[\![\exists x\!:\!T.\,Px]\!]$ is a pair $\langle y, pf\rangle$ such that y is in $[\![T]\!]$ and pf is a proof of $y \in T$. Since call-by-name reduction prevents the proof part from being evaluated, dead code does not affect program performance. Proposals for improving the readability of extracts include dead code elimination (Hafızogulları & Kreitz, 1998) and the use of set types for existential quantification (Caldwell, 1997). Similar ideas for the COQ system were made in (Paulin-Mohring, 1989; Paulin-Mohring & Werner, 1993).

1.2 The ACAp System

The *Automated Complexity Analysis Prototype* (ACA*p*) is an experimental system designed and implemented by the author to automatically determine the time complexity of NUPRL extracts. It is based on the call-by-name semantics of NUPRL version 4.2 and uses static program analysis techniques to infer the number of reduction steps required to reduce instantiations of a given open term to canonical form. The current implementation is limited to first-order functions, lists, and a subclass of higher-order functions, but we will outline in Section 6.1 how we plan to extend the capabilities of the system to deal with general higher-order functions. The main system is written in NUPRL's ML language and consists of about 3,000 lines of code. Additional modules are implemented as MATHEMATICA packages and account for about 300 lines of code.

The system takes a NUPRL term as input and tries to determine its time complexity in terms of user-specified parameters. First, the *symbolic evaluator* generates a raw complexity expression through abstract interpretation of the input program. At the same time, the *recurrence generator* translates recursive calls into recurrence equations, which are stored separately for the *recurrence solver*. Once symbolic eval-

uation is complete, the *simplifier* rewrites the raw expression in human-readable form while making repeated calls to the recurrence solver to substitute occurring recurrence variables by closed expressions (cf. Figure 1). The recurrence solver contains a collection of individual *rsolver modules* that apply to particular classes of recurrence equations. The rsolvers are repeatedly swept over the list of unsolved recurrence equations until no further progress is made. If unsolved recurrences remain, they are reported along with the partially unresolved complexity expression.

The answer given by the system is not "provably correct" in the sense of automatic theorem proving but is subject to the usual design and implementation errors. This is not a principal limitation, though, as the data gathered during the analysis could be used to construct a formal proof that asserts the running time behavior reported by the ACAp system. The implementation details of this translation will be a topic of future investigation.

Although our system is just a prototype, it is already quite useful in its limited domain of lists and primitive recursion. By using meta level heuristics and approximations when necessary, the system can analyze a large class of programs arising in practical program synthesis, even though complexity analysis of general recursive programs is non-computable. We should also note that the system is not targeted at outperforming theoreticians analyzing a newly discovered algorithm, but at automating the standard "bulk" complexity analyses that are common in program synthesis.

1.3 Outline

The remaining of this paper is organized as follows: Section 2 formally describes our definition of time complexity for NUPRL's term languages. This definition is the basis for our calculus of symbolic evaluation described in Section 3. Section 4 outlines the simplifier and the recurrence solver of the ACAp system. Section 5 illustrates the combined use of NUPRL and the ACAp system for program development by synthesizing and analyzing a solution to the maximum segment sum problem and by analyzing a functional program to compute the minimum of a list by sorting.

2 Time Complexity of Functional Programs

Common formal definitions of time complexity for imperative programs are based on the Turing machine or the random access machine model. For first-order functions, these models relate the size of the input to the number of transitions made by the machine. Higher-order functionals require the use of oracle machines that encode functions passed as input in an appropriate oracle. Even though this imperative notion of time complexity is tightly bound to the particular machine model being used, it is well-known that RAMs and generalized Turing machines such as k-tape machines yield sufficiently similar results in that they can simulate each other within a polynomial factor in time and a constant factor in space. In particular, the class P of *feasible programs* running in polynomial time is independent of the machine model being used, guaranteeing a certain robustness to reasonable modifications.

Defining an accurate cost model for a functional language is a considerably harder business, as different reduction strategies yield dramatically different performances, and cost measures for an abstract machine might not necessarily reflect the real world (Lawall & Mairson, 1996). In fact, no satisfactory implementation-independent characterization of evaluation cost has been found yet. A faithful cost model would need to relate the cost of one primitive reduction step of the abstract machine to the actual cost of the implementation of this step (Greiner, 1997). As is customary, we decided to ignore this aspect for our preliminary analysis and based the cost model solely on the number of steps given by the abstract call-by-name operational semantics of the current NUPRL evaluator (Constable *et al.* , 1986). The new METAPRL system (Hickey, 2000) will feature a highly efficient evaluator using sharing and explicit substitution whose semantics should be fine-grained enough to faithfully classify the class of feasible programs even without translating from the abstract machine to the actual implementation.

We define the time complexity of a term t with respect to semantics $\mathcal{S}$ as the number of primitive reduction steps required by $\mathcal{S}$ to reduce t to canonical form. Throughout this paper, we will use the profiling semantics shown in Figure 2, where

$$t \downarrow w \text{ (in } n)$$

denotes that term t reduces to canonical form w in at most n reduction steps. As a convention, we will call w the *result* and n the *complexity* of t. To avoid ambiguities between object language and meta language, we will often refer to terms representing complexities as *expressions*.

Given above definitions, the complexity of $1+(1+1)$ is 2, whereas the complexity of $\lambda x.x + 1$ is 0. Obviously, the latter result does not reflect that when given a function f, we are really interested in the number of reduction steps required to compute $f(t)$ for any argument t. Thus, we rephrase our analysis by introducing *meta variables* denoting arbitrary terms in our term language. For all practical matters, these meta variables are simply free variables.

Returning to the example above, we see that the complexity of $((\lambda x.x + 1)\ m)$ is $\mathtt{time}(m) + 2$, where $\mathtt{time}(m)$ is an expression denoting the time complexity of the unknown argument m. As another example, let $power == \lambda k.\mathtt{ind}(k;\ 1;\ i,z.z * x)$. Again, the complexity of *power* is 0, but the complexity of $(power\ m)$ is $2 + 2m + m\,\mathtt{time}(x) + \mathtt{time}(m)$.

2.1 Higher-Order Functions

Several problems arise when applying a reduction step count to higher-order functions. Consider the simple functional

$$twice == \lambda f.\, \lambda x.\, f(fx)$$

of type $(\alpha \to \alpha) \to \alpha \to \alpha$. What is the computational complexity we should assign to this term? Is *twice* an operator, mapping a function of type $\alpha \to \alpha$ to another function of the same type, or is it a function with two arguments of type $\alpha \to \alpha$ and α, respectively, mapping to α? Since currying justifies both views equally, one

$$\text{(canon)} \quad w \downarrow w \ (\text{in } 0) \quad (w \text{ canonical term})$$

$$\text{(ap)} \quad \frac{f \downarrow \lambda x.b \ (\text{in } n_1), \quad b[u/x] \downarrow w \ (\text{in } n_2)}{\texttt{ap}(f; u) \downarrow w \ (\text{in } n_1 + n_2 + 1)}$$

$$\text{(spread)} \quad \frac{p \downarrow \langle u, v\rangle \ (\text{in } n_1), \quad t[u/a, v/b] \downarrow w \ (\text{in } n_2)}{\texttt{spread}(p; a, b.t) \downarrow w \ (\text{in } n_1 + n_2 + 1)}$$

$$\text{(decide)} \quad \frac{p \downarrow \texttt{inl}(t) \ (\text{in } n_1), \quad u[t/a] \downarrow w \ (\text{in } n_2)}{\texttt{decide}(p; a.u; b.v) \downarrow w \ (\text{in } n_1 + n_2 + 1)} \quad (\text{ditto } \texttt{inr})$$

$$\text{(arith)} \quad \frac{u \downarrow k_1 \ (\text{in } n_1), \quad v \downarrow k_2 \ (\text{in } n_2)}{\texttt{add}(u; v) \downarrow k_1 + k_2 \ (\text{in } n_1 + n_2 + 1)} \quad (\text{ditto } \texttt{sub}, \texttt{mul}, \texttt{div}, \texttt{rem})$$

$$\text{(comp)} \quad \frac{p \downarrow k_1 \ (\text{in } n_1), \ q \downarrow k_2 \ (\text{in } n_2), \ k_1 < k_2, \ u \downarrow w \ (\text{in } n_3)}{\texttt{less}(p; q; u; v) \downarrow w \ (\text{in } n_1 + n_2 + n_3 + 1)} \quad (\text{ditto } \texttt{int\_eq})$$

$$\text{(indbase)} \quad \frac{p \downarrow 0 \ (\text{in } n_1), \quad b \downarrow w \ (\text{in } n_2)}{\texttt{ind}(p; b; i,z.f) \downarrow w \ (\text{in } n_1 + n_2 + 1)}$$

$$\text{(indstep)} \quad \frac{p \downarrow k > 0 \ (\text{in } n_1), \quad u[k/i, \ \texttt{ind}(k-1; b; i,z.f)/z] \downarrow w \ (\text{in } n_2)}{\texttt{ind}(p; b; i,z.f) \downarrow w \ (\text{in } n_1 + n_2 + 1)}$$

$$\text{(listbase)} \quad \frac{l \downarrow \texttt{[]} \ (\text{in } n_1), \quad b \downarrow w \ (\text{in } n_2)}{\texttt{listind}(l; b; h,t,z.f) \downarrow w \ (\text{in } n_1 + n_2 + 1)}$$

$$\text{(liststep)} \quad \frac{l \downarrow u :: v \ (\text{in } n_1), \ f[u/h, \ v/t, \ \texttt{listind}(v; b; h,t,z.f)/z] \downarrow w \ (\text{in } n_2)}{\texttt{listind}(l; b; h,t,z.f) \downarrow w \ (\text{in } n_1 + n_2 + 1)}$$

$$\text{(abs)} \quad \frac{\texttt{op}(\bar{u}) == \phi(\bar{u}), \quad \phi(\bar{u}) \downarrow w \ (\text{in } n)}{\texttt{op}(\bar{u}) \downarrow w \ (\text{in } n + 1)} \quad (\texttt{op} \text{ user definition})$$

Fig. 2. Profiling semantics of the call-by-name evaluator

might argue that the complexity expression for *twice* should convey information for either case. For example, a pair (c_1, c_2) could denote that *twice* applied to some term t reduces to some function g in c_1 steps, and g applied to some term t' reduces to some value w in c_2 steps. The problem with this approach is that we cannot express how c_2 might potentially depend on t. In particular, in the case of *twice*, c_2 is *not* just about twice the complexity of f: applying *twice* to $f_1 == \lambda y.0$ and $f_2 == \lambda y.y * y * y$ shows that (*twice* f_1 m) and (f_1 m) have about the same complexity, whereas (*twice* f_2 m) takes about three times as many reduction steps as (f_2 m).

With our goal of user-directed complexity analysis in mind, we decided to follow a more pragmatic approach by demanding complexity information for higher-order input terms. Central to our calculus is a new term $\texttt{cpx}(n; t)$ that reduces to t in n steps (cf. Figure 3). Using this new term, we can abstract from the value denoted by a term while retaining its time complexity. In particular, this allows us to define *abstract functions* $f == \lambda x.\texttt{cpx}(n_x; t_x)$ that represent the class of functions with

$$\text{(cpx)} \quad \frac{t \downarrow w \ (\text{in } n_1)}{\texttt{cpx}(n;t) \downarrow w \ (\text{in } n + n_1)}$$

Fig. 3. Rule for complexity abstraction

the same computational behavior as f. For example, the result of *twice* applied to a linear function like $\lambda y.\texttt{cpx}(y;y)$ has complexity $O(2m)$.

Our ability to define abstract functions puts us in a position to demand that *all free variables be of type integer* in order to simplify our calculus. For technical reasons we will outline in Section 3, the current implementation of ACAp also requires that all computationally relevant arguments of functional inductive terms be of type integer. At first, this requirement might appear as being too strong a restriction to permit the analysis of any interesting programs. Our experience has shown, however, that many useful algorithms synthesized by NUPRL do adhere to this assumption: Although internally generated functions are generally of higher type than the resulting program, the computationally relevant arguments of these internal functions are often enough of simple type. Additionally, higher types are likely to be confined within the proof part of the program, which is irrelevant to the analysis and ignored by the system.

Given the current state of the prototype, abstract functions provide a convenient way to specify higher-order inputs, and the user can easily run several analyses for different function classes. We would like to emphasize, though, that the integer variable assumption is *not* a conceptual limitation of our calculus. Instead, we believe that abstraction via `cpx` terms leads to a natural description of higher-order functions in terms of their first-order components. Like inductive terms, free variables of higher type may be replaced by their "computational skeleton" to introduce first-order quantities that we can reason about. Thus, as we will sketch in Section 6.1, the main problem of dealing with higher-order functions does not arise from the calculus but from the recurrences being generated by higher-order inductive terms.

2.2 Lists

Lists are often the only built-in data structure of functional programming languages and thus have a large influence on the performance of non-numerical algorithms. Call-by-name evaluation of lists, though, has some peculiarities of its own: consider the function

$$incr == \lambda l.\texttt{listind}(l;\texttt{[]};h,t,z.(h+1) :: z)$$

of type $\mathbb{Z}$ `list` $\to \mathbb{Z}$ `list`. NUPRL reduces $(incr\ 1 :: 2 :: \texttt{[]})$ to $2 :: (incr\ 2 :: \texttt{[]})$ rather than $2 :: 3 :: \texttt{[]}$, because '::' is a canonical operator. The call-by-name semantics prevent full evaluation to strong normal form, which defers part of the cost normally associated with the construction of the list to its destruction.

Analogous to abstract functions, we introduce the notion of *abstract lists* to specify generic lists. The canonical term $\texttt{lcpx}(n;e;k)$ denotes the class of lists of

$$\text{(lbase)}\quad \frac{l \downarrow \texttt{lcpx}(n; e; 0)\ (\text{in } n_1), \quad b \downarrow w\ (\text{in } n_2)}{\texttt{listind}(l; b; h,t,z.f) \downarrow w\ (\text{in } n_1 + n_2 + 1)}$$

$$\text{(lcpx)}\quad \frac{\begin{array}{c} l \downarrow \texttt{lcpx}(n; e; k)\ (\text{in } n_1), \\ f[e/h,\ \texttt{cpx}(n; \texttt{lcpx}(n; e; k-1))/t, \\ \texttt{listind}(\texttt{lcpx}(n; e; k-1); b; h,t,z.f)/z] \downarrow w\ (\text{in } n_2) \end{array}}{\texttt{listind}(l; b; h,t,z.f) \downarrow w\ (\text{in } n_1 + n_2 + 1)}$$

Fig. 4. List induction over abstract lists

length k with *spine complexity* n and *generic element* e. The spine complexity is the number of reduction steps required to reduce the tail of the list to the next '::' term, e.g. the spine complexity of $\texttt{ind}(k; \texttt{[]}; i,z.\, i :: \texttt{cpx}(a; z))$ is $a + 1$. Each element in the list is assumed to have the same type and complexity as the generic element e, which is usually a combination of `cpx` terms. Intuitively, one can think of $\texttt{lcpx}(n; e; k)$ as being semantically equivalent to $e :: \texttt{cpx}(n; \texttt{lcpx}(n; e; k-1))$. The new list induction rule for abstract lists is shown in Figure 4.

We illustrate the use of abstract lists with a few examples. The function

$$\mathit{fold} == \lambda L.\lambda x.\texttt{listind}(L;\ x;\ h,t,z.h\ z)$$

successively applies a list of functions onto a single element. Then $(\mathit{fold}\ \texttt{lcpx}(a; \lambda x.\texttt{cpx}(\texttt{time}(x)+ b; \texttt{Ax}); m)\ e)$ reduces to canonical form in $3 + 2m + am + bm + \texttt{time}(e)$ steps. As another example, if

$$\mathit{sumprod} == \lambda L.\texttt{listind}(L;\ 0;\ h_1,t_1,z_1.z_1 + \texttt{listind}(t_1;\ 1;\ h_2,t_2,z_2.h_2 * z_2)),$$

then $(\mathit{sumprod}\ \texttt{lcpx}(a; e; m))$ reduces to canonical form in $2 + 2m + m^2 + (3 + m)am/2 + (m-1)m\,\texttt{time}(e)/2$ steps. Finally, let

$$\begin{aligned} \mathit{silly} \quad == \quad & \lambda L.\texttt{listind}(\texttt{listind}(L; \texttt{[]}; \\ & \qquad h1,t1,z1.\texttt{listind}(z1; \texttt{[]}; h2,t2,z2.h1 :: z2)); 0; h3,t3,z3.h3 + z3), \end{aligned}$$

where the outermost list induction simply adds up all the elements of the list in order to measure the full cost of computing with the lazy list returned by the inner inductions. Analysis of $(\mathit{silly}\ \texttt{lcpx}(a; e; m))$ with the ACAp system yields $3 + 2m + am$, a result that is not so easily obtained by hand. Careful examination reveals that the list returned by the inner inductions is always the empty list, which explains why the complexity expression does not depend on the element representative e.

Both `cpx` and `lcpx` terms may be used by the user to describe the input to the algorithm under investigation. Additionally, the system itself introduces these terms for representatives of inductive terms and as results of inductions of type α `list` (cf. Section 3.3).

$$\text{(apv)}\quad \frac{f \downarrow \lambda x.b\ (\text{in } n_1),\quad u \downarrow v\ (\text{in } n_2),\quad b[v/x] \downarrow w\ (\text{in } n_3)}{\mathtt{ap}(f;u) \downarrow w\ (\text{in } n_1+n_2+n_3+1)}$$

$$\text{(spreadv)}\quad \frac{p \downarrow \langle p_1,p_2\rangle\ (\text{in } n_1),\ p_1 \downarrow v_1\ (\text{in } n_2),\ p_2 \downarrow v_2\ (\text{in } n_3),\ t[v_1/a,v_2/b] \downarrow w\ (\text{in } n_4)}{\mathtt{spread}(p;a,b.t) \downarrow w\ (\text{in } n_1+n_2+n_3+n_4+1)}$$

$$\text{(indstepv)}\quad \frac{p \downarrow k>0\ (\text{in } n_1),\ \mathtt{ind}(k-1;b;i,z.f) \downarrow v\ (\text{in } n_2),\ u[k/i,\ v/z] \downarrow w\ (\text{in } n_3)}{\mathtt{ind}(p;b;i,z.f) \downarrow w\ (\text{in } n_1+n_2+n_3+1)}$$

Fig. 5. Profiling semantics for call-by-value evaluation

2.3 Alternative Semantics

Even though the current NUPRL system supports only call-by-name evaluation of terms, the ACAp system can easily be adapted to handle other reduction strategies as well. As an example, we implemented a call-by-value evaluation scheme, for which Figure 5 shows some of the rules.

3 Symbolic Evaluation

A program synthesized by NUPRL is a closed term p of some implicit type $\tau = \tau_1 \to \tau_2 \to \cdots \to \tau_n$. Since τ_n might be an arrow type, typing of p is inherently ambiguous and reflects the user's view of the program (cf. Section 2.1). The complexity of p is defined relative to arguments a_i of type τ_i $(i = 1, \ldots, n-1)$ and will be expressed in terms of the free variables contained in these arguments.

Analysis proceeds by evaluating the term $(p\ a_1\ \cdots\ a_{n-1})$ using the *symbolic evaluation rules* listed in the subsequent sections. These rules are a conservative extension of the standard NUPRL semantics to open or *symbolic* terms involving free variables. The symbolic reduction rule for a term $\mathtt{op}(p_{\bar{v}};\bar{t})$ with principal argument $p_{\bar{v}}$ and free variables $\bar{v}$ specifies some (possibly symbolic) result $w_{\bar{v}}$ and some complexity expression $n_{\bar{v}}$ such that

$$\mathtt{op}(p_{\bar{u}};\bar{t}) \downarrow w_{\bar{u}}\ (\text{in } n_{\bar{u}}) \tag{1}$$

holds for all type-correct instantiations $\bar{u}$ of $\bar{v}$. Intuitively, symbolic evaluation aims at reducing non-canonical terms t even when the principal argument is unknown. It is therefore unreasonable to demand that the result t' of t should be "simpler" than t, as t' has to capture the evaluation paths for all admissible instantiations of t. This gain in information at the expense of simple term structure is justified by our indifference about the actual value of the computation.

The following sections group the standard semantics of Figure 2 into rules relating to arithmetic, case analysis, recursion, and list processing and address the adjustments needed in each situation. Rules not covered below, such as function application, need not be modified for symbolic evaluation because of the integer variable assumption.

$$\text{(var)} \quad x \downarrow x^* \ (\text{in } \texttt{time}(x)) \quad (x \text{ is variable of type } \mathbb{Z})$$

$$\text{(arith)} \quad \frac{k_1 \downarrow k_1' \ (\text{in } n_1), \quad k_2 \downarrow k_2' \ (\text{in } n_2)}{\texttt{add}(k_1; k_2) \downarrow \texttt{add}^*(k_1'; k_2') \ (\text{in } n_1 + n_2 + 1)} \quad (\text{ditto } \texttt{sub}, \texttt{mul}, \texttt{div})$$

Fig. 6. Symbolic arithmetic rules

3.1 Symbolic Arithmetic

A free variable v, by assumption, represents an unknown integer which might be of arbitrary complexity. We use two new canonical terms, v^* and $\texttt{time}(v)$, to denote the value and the time complexity of an admissible instantiation of v, respectively. Symbolic evaluation will generate many intermediate `time` expressions for many different program variables, but only those referring to free variables will persist in the resulting complexity expression returned to the user. The term `Ax` is frequently used in `cpx` expressions to denote an arbitrary integer in canonical form, but any canonical term would serve the same purpose.

For each arithmetical operator `op`, we introduce a new canonical term $\texttt{op}^*$ that represents an evaluated term (cf. Figure 6). Note that it is irrelevant that $(a+1)+2$ reduces to $(a^* +^* 1) +^* 2$ whereas $a + (1 + 2)$ reduces to $a^* +^* 3$; all that matters is that both reductions require $\texttt{time}(a) + 2$ steps and that both results denote the same value when identifying $+^*$ with $+$.

3.2 Symbolic Cases

The bottom-up style of the NUPRL semantics requires that the symbolic result of a case split entail both possible computation branches. Let t be a comparison $\texttt{less}(k_1; k_2; t_l; t_r)$ with at least one symbolic principal argument and assume that $k_i \downarrow k_i' \ (\text{in } n_i)$, $t_l \downarrow t_l' \ (\text{in } n_l)$ and $t_r \downarrow t_r' \ (\text{in } n_r)$ hold. Without further information on k_1 and k_2, the best upper bound on the complexity of t is $k_1 + k_2 + \max(n_l, n_r) + 1$. We thus define the result of t to be a new *indeterminate term* $\texttt{indet}(t_l'; t_r')$, which serves as a placeholder for instantiations of both t_l' and t_r' (cf. Figure 7). Note that even if $n_l > n_r$ were valid for all instantiations, it is not permissible to discard t_r', as the simple example

$$(\texttt{less}(p; q; \texttt{cpx}(10; \lambda x.x); \lambda x.\texttt{cpx}(20; x)) \ 1)$$

shows.

The ambiguity introduced by symbolic case splits cannot be resolved but will percolate to the root of the computation tree. Any operation on an indeterminate term $\texttt{indet}(u; v)$ is performed on both subterms u and v, and the complexity of this operation will be the maximum of the complexities of the subterm operations. Thus, indeterminate terms are neither canonical nor non-canonical but *transparent*, for evaluation is channeled into the subterms. Transparency affects all symbolic reduction rules, but it is possible to subsume all required alterations into meta rule

$$\text{(comp)}\quad \frac{k_1 \downarrow k_1' \ (\text{in } n_1),\quad k_2 \downarrow k_2' \ (\text{in } n_2),\quad \texttt{indet}(u;v) \downarrow w \ (\text{in } n_3)}{\texttt{less}(k_1;k_2;u;v) \downarrow w \ (\text{in } n_1+n_2+n_3+1)} \quad (\text{ditto } \texttt{int\_eq})$$

$$\text{(indet)}\quad \frac{p \downarrow \texttt{indet}(p_l;p_r) \ (\text{in } n_1),\ \texttt{op}(p_l;\bar{u}) \downarrow w_l \ (\text{in } n_l),\ \texttt{op}(p_r;\bar{u}) \downarrow w_r \ (\text{in } n_r)}{\texttt{op}(p;\bar{u}) \downarrow \texttt{indet}(w_l;w_r) \ (\text{in } n_1 + \texttt{tmax}(n_l;n_r))} \quad (\text{any } \texttt{op})$$

Fig. 7. Symbolic case split rules

$$\sum_j \texttt{tmax}_{i_j}(p_j;q_j) = \sum_{k \in \{i_1,i_2,\dots\}} \max\left(\sum_{j \mid i_j=k} p_j,\ \sum_{j \mid i_j=k} q_j\right)$$

Fig. 8. Semantics of `tmax` terms

(indet) shown in Figure 7. This meta rule supersedes any symbolic reduction rule whenever the principal argument p is indeterminate.

A new expression `tmax`, for *total maximum*, prevents that bounds obtained in this fashion become unnaturally loose. Let $\texttt{ifless}(k_1;k_2;u;v)$ be an abstraction for $\texttt{decide}(\texttt{less}(k_1;k_2;\texttt{inl}(\texttt{Ax});\texttt{inr}(\texttt{Ax}));u;v)$ and consider the term

$$(\texttt{isless}(p;q;\texttt{cpx}(10;\lambda x.\texttt{cpx}(1;\texttt{Ax}));\texttt{cpx}(1;\lambda x.\texttt{cpx}(10;\texttt{Ax}))) \ 0),$$

which reduces in $\texttt{time}(p) + \texttt{time}(q) + \texttt{tmax}_1(10;1) + \texttt{tmax}_1(1;10) + 3$ steps for all admissible terms p and q. If we interpreted `tmax` as the conventional maximum function, the bound would evaluate to $\texttt{time}(p) + \texttt{time}(q) + 23$ even though the actual program reduces in $\texttt{time}(p) + \texttt{time}(q) + 14$ steps—an excess of 9 steps in our analysis! The semantics of $\texttt{tmax}_i$ thus capture the notion that reduction steps of different computation paths cannot be mixed: the total maximum of a group of $\texttt{tmax}_i$ expressions with identical labels i, where i is a number that is uniquely assigned to each comparative term, is the maximum of the sum of their first and the sum of their second arguments (cf. Figure 8).

On occasions, this bound might still be somewhat conservative, for example if the proposition tested is a tautology or depends on previous case splits. Graph algorithms in particular often involve the following kind of argument:

> "Each of the n loop iterations might perform up to n operations, suggesting an $O(n^2)$ algorithm; in fact, there are only two different kinds of operation and n entities to perform these operations on, so the total running time must be $O(2n)$."

An accurate analysis of nested case splits is generally undecidable, but future heuristics might be able to model above reasoning in selected important domains.

3.3 Primitive Recursion

Symbolic primitive recursion is particularly difficult to analyze because an infinite number of possible computation paths are confined into one closed expression. Given an inductive term $t == \texttt{ind}(k;b;i,z.f)$, we can evaluate the base case b to obtain

the time complexity of t for $k = 0$. For the step case, however, the time complexity of the free variable z is unknown, and it cannot be determined by unrolling the recursion when the principal argument is symbolic.

Our main idea to tackle symbolic recursion is to identify a symbolic inductive term t with its "computational skeleton" consisting of nested `cpx` terms. The purpose of the skeleton is to abstract from the actual value being computed while retaining the computational behavior of t in a non-recursive form. During symbolic evaluation, the `cpx` terms introduce names for the unknown time complexities of t so that, in separate steps, we can derive and solve difference equations for these quantities in order to obtain closed expressions. For example, the skeleton of any term t of type $\mathbb{Z} \times \mathbb{Z}$ may be represented by

$$\mathtt{cpx}(\rho; \langle \mathtt{cpx}(\rho_1; \mathtt{Ax}), \mathtt{cpx}(\rho_2; \mathtt{Ax}) \rangle),$$

where ρ, ρ_1, and ρ_2 are certain complexity expressions that depend on the free variables in t. In other words, the term t will reduce to some canonical form $\langle t_1, t_2 \rangle$ in ρ steps, where t_1 and t_2 need not be canonical but will reduce to canonical form in ρ_1 and ρ_2 steps, respectively. Similarly, a term u of type $\mathbb{Z} \to \mathbb{Z}$ may be abstracted to

$$\mathtt{cpx}(\rho; \lambda x.\mathtt{cpx}(\rho_{\mathtt{a}}(x); \mathtt{Ax})),$$

where $\rho_{\mathtt{a}}(x)$ is now a functional expression that depends not only on the free variables in u but also on (the time complexity of) x.

In general, given a term t of type α, we define the *representative* of t to be the term

$$\mathrm{rep}_m(t) == \mathtt{cpx}(\rho(m); R(\alpha, \epsilon)),$$

where ϵ is the empty sequence and R recursively builds the skeleton as defined by

$$R(\alpha, d) := \begin{cases} \lambda x.\mathtt{cpx}(\rho_{d\cdot\mathtt{a}}(\bar{u}; x); R(\beta, d\cdot\mathtt{a})) & \text{if } \alpha = \mathbb{Z} \to \beta \\ \langle \mathtt{cpx}(\rho_{d\cdot 1}(\bar{u}); R(\beta, d\cdot 1)), & \\ \quad \mathtt{cpx}(\rho_{d\cdot 2}(\bar{u}); R(\gamma, d\cdot 2)) \rangle & \text{if } \alpha = \beta \times \gamma \\ \mathtt{lcpx}(\rho_{d\cdot\mathtt{t}}(\bar{u}); R(\beta, d\cdot\mathtt{e}); \rho_{d\cdot\mathtt{s}}(\bar{u})) & \text{if } \alpha = \beta\ \mathtt{list} \\ \mathtt{Ax} & \text{if } \alpha = \mathbb{Z} \end{cases}.$$

So far, the ρ_d occuring in the representative are mere symbols, for which concrete expressions will have to be obtained separately. We refer to character sequence d as the *descriptor*, since it describes the position of the subterm that ρ_d characterizes. There is no deeper meaning behind above naming convention over alphabet $\{\mathtt{a}, \mathtt{1}, \mathtt{2}, \mathtt{e}, \mathtt{s}, \mathtt{t}\}$ other than aiding the user in identifying the origin of variables occuring in unsimplified complexity expressions.

If all ρ_d are known, the term $t\colon \alpha$ and its representative $\mathrm{rep}(\alpha)$ are interchangeable without affecting the time complexity of their context. Now in order to evaluate an inductive term $t_k == \mathtt{ind}(k; b; i, z.f)$ of type α, we compute its representative $r = \mathrm{rep}_m(\alpha)$ and evaluate $r[k/m]$ in lieu of t_k. This reduction will yield a complexity expression containing all or some of the symbols ρ_d of r. To derive concrete expressions for each ρ_d, we unfold the step case f once to infer a functional de-

$$\text{(ind)}\quad \frac{k \downarrow k' \text{ (in } n_1),\quad \mathrm{rep}_{k'}(\alpha) \downarrow w \text{ (in } n_2),\quad \mathrm{getrecr}(\rho_\epsilon(m); b; f[m/i, \mathrm{rep}_{m-1}(\alpha)/z])}{\mathtt{ind}(k; b; i,z.f)\colon \alpha \downarrow w \text{ (in } n_1 + n_2)}$$

Fig. 9. Symbolic induction rule

pendency between $\rho_d(m)$ and $\rho_d(m-1)$. By assumption, $t_m \downarrow w$ (in $\rho(m)$) and $t_{m-1} \downarrow w'$ (in $\rho(m-1)$) for canonical m and $m-1$, where both $\rho(m)$ and $\rho(m-1)$ are unknown. But obviously

$$\begin{aligned}\rho(m) &= \mathtt{time}(t_m)\\ &= 1 + \mathtt{time}(f[m/i, t_{m-1}/z])\\ &= 1 + \mathtt{time}(f[m/i, \mathtt{cpx}(\rho(m-1), t')/z])\\ &= 1 + \varphi(\rho(m-1))\end{aligned}$$

for some function φ that is easily obtained by evaluating $f[m/i, \mathtt{cpx}(\rho(m-1), t')/z]$. Together with $\rho(0) = \mathtt{time}(b)$, this yields a recurrence relation

$$\rho_d(m) = \begin{cases} \varphi(m, \rho_d(m-1), \bar{y}) & \text{if } m > 0 \\ \psi(\bar{x}) & \text{if } m = 0 \end{cases},$$

where m denotes the value of k and $\bar{x}$, $\bar{y}$ are the free variables of b and $i,z.f$, respectively.

The careful reader may have observed that above derivation for $\rho(m)$ involves an unknown term t' denoting the result of reducing t_{m-1}. If t is of type $\mathbb{Z}$, then t' is some canonical integer which we will replace by `Ax`. Otherwise, the representative $\mathrm{rep}(\alpha)$ involves other unknown variables ρ_d that describe the complexity of the subterms of t', and for which concrete expressions must be inferred (cf. Figure 10).

In particular, if t is of type $\beta \times \gamma$, then t' is of the form $\langle t_1 : \beta; t_2 : \gamma \rangle$ with (at least) unknown variables ρ_{d1} and ρ_{d2}. By re-iterating above procedure for t_1 and t_2, we obtain recurrences for ρ_{d1}, ρ_{d2} and new results t'_1, t'_2, which are again subject to further analysis.

If t is of type $\beta \to \gamma$, then t' is of the form $\lambda x : \beta.u : \gamma$ involving (at least) unknown variable $\rho_{d\mathtt{a}}$. Again, we re-iterate by evaluating u to derive a recurrence equation for $\rho_{d\mathtt{a}}$ and some result u'. Note that $\rho_{d\mathtt{a}}$ depends on x; for each arrow type, i.e. for each subscript `a`, the corresponding recurrence variable will involve an additional parameter. Since variable x is free in u, the integer variable assumption applies and limits the types of inductions supported. We outline in Section 6.1 how this restriction can be relaxed.

If t is of type β `list`, we replace t' by another `lcpx` term. The function getlist (cf. Figure 11) will decompose the remaining parts of b and f into the elements and the tails of the new list while keeping track of the spine complexity $\rho_{d\mathtt{t}}$. All elements that might become members of the list will be wrapped in an `indet` term that will become the new element representative. Finally, a recurrence relation for the length of the new list $\rho_{d\mathtt{s}}$ is generated.

The representative abstracts from the actual value of the induction by replacing

$$\text{(lam)}\quad \frac{b_d \downarrow \lambda x.b_{d\mathtt{a}}\ (\text{in } n_0),\quad f_d \downarrow \lambda x.f_{d\mathtt{a}}\ (\text{in } n_m)}{\rho_d(0,\bar{p}) := n_0,\quad \rho_d(m,\bar{p}) := n_m,\quad \text{getrecr}(\rho_{d\mathtt{a}}(m,\bar{p},x); b_{d\mathtt{a}}; f_{d\mathtt{a}})}$$

$$\text{(pair)}\quad \frac{b_d \downarrow \langle b_{d1}, b_{d2}\rangle\ (\text{in } n_0),\quad f_d \downarrow \langle f_{d1}, f_{d2}\rangle\ (\text{in } n_m)}{\rho_d(0,\bar{p}) := n_0,\quad \rho_d(m,\bar{p}) := n_m,\quad \text{getrecr}(\rho_{di}(m,\bar{p}); b_{di}; f_{di}),\ i=1,2}$$

$$\text{(list)}\quad \frac{b_d \downarrow e_{b0} :: b'_d\ (\text{in } n_0),\quad f_d \downarrow e_{f0} :: f'_d\ (\text{in } n_m)}{\begin{array}{c}\rho_d(0,\bar{p}) := n_0,\quad \rho_d(m,\bar{p}) := n_m,\\ e_b, e_f := \text{getlist}(\rho_d(m,\bar{p}); e_{b0} :: b'_d; e_{f0} :: f'_d),\quad \text{getrecr}(\rho_{d\mathtt{e}}(m,\bar{p}); e_b; e_f)\end{array}}$$

$$\text{(int)}\quad \frac{b_d \downarrow b' : \mathbb{Z}\ (\text{in } n_0),\quad f_d \downarrow f' : \mathbb{Z}\ (\text{in } n_m)}{\rho_d(0,\bar{p}) := n_0,\quad \rho_d(m,\bar{p}) := n_m}$$

Fig. 10. Recurrence generation 'getrecr'

$$\frac{\begin{array}{c}b' \downarrow e_{b1} :: b''\ (\text{in } n_{b1}),\ b'' \downarrow e_{b2} :: b'''\ (\text{in } n_{b2}),\ \ldots,\ b^{(k)} \downarrow [] \text{ or } \mathtt{lcpx}(n_b; e_b; k')\ (\text{in } n_{bk}),\\ f' \downarrow e_{f1} :: f''\ (\text{in } n_{f1}),\ f'' \downarrow e_{f2} :: f'''\ (\text{in } n_{f2}),\ \ldots,\ f^{(\ell)} \downarrow [] \text{ or } \mathtt{lcpx}(n_f; e_f; \ell')\ (\text{in } n_{f\ell})\end{array}}{\begin{array}{c}\rho_{d\mathtt{s}}(0) := k + k',\ \rho_{d\mathtt{s}}(m) := \ell + \ell',\\ \rho_{d\mathtt{t}}(0) := \mathtt{tmax}(n_{b1}; \ldots; n_{bk}; n_b),\ \rho_{d\mathtt{t}}(m) := \mathtt{tmax}(n_{f1}; \ldots; n_{f\ell}; n_f),\\ \text{getlist}(\rho_d; e_{b0} :: b'; e_{f0} :: f') \to \mathtt{indet}(e_{b0}; \ldots; e_{bk-1}; e_b), \mathtt{indet}(e_{f0}; \ldots; e_{f\ell-1}; e_f)\end{array}}$$

Fig. 11. List analysis 'getlist'

all integer results by the constant `Ax`. This loss of information will not affect the analysis unless the inductive term is the principal argument of another inductive term. Our experience suggests, however, that this is a very rare instance; usually, nested recursion corresponds to well-nested loops so that the inductive term is included within the step case of another inductive term.

Some examples will illustrate the use of these rules. Let

$$\mathit{power} == \mathtt{ind}(m; \lambda x.1; i,z.\lambda x.(z\ i) * 2).$$

The complexity of *power* is $\rho(m) \equiv \mathtt{time}(m)+1$, for the actual computation of 2^m is deferred until a dummy argument is passed to the resulting function. This is a rather unusual way to implement *power*, but NUPRL indeed synthesizes the function this way. We can see that $\rho_{\mathtt{a}}$ depends on m whereas parameter x is ignored; the exact relationship is $\rho_{\mathtt{a}}(m,x) = \rho_{\mathtt{a}}(m-1,0)+3$, $\rho_{\mathtt{a}}(0,x) = 0$, hence the representative of *power* is $\mathtt{cpx}(\mathtt{time}(m)+1; \lambda x.\mathtt{cpx}(3*m; \mathtt{Ax}))$.

As another example, the function

$$\mathit{table} == \lambda f.\lambda k.\mathtt{ind}(k; [\,]; i,z.(f\ i) :: z).$$

creates a list of elements $f(k) :: \cdots :: f(1)$. For some function f that runs in constant time, the complexity of $(\mathit{table}\ \lambda x.\mathtt{cpx}(n; \mathtt{Ax}))\ m$ is $3 + \mathtt{time}(m)$—lazy evaluation will defer most of the work.

Finally, let

$$\mathit{hyper} == \lambda k.\mathtt{ind}(k; \lambda x.x; i,z.\lambda x.z((z\ x)+1))$$

$$\text{(listind)}\quad \frac{\begin{array}{c} l \downarrow \texttt{lcpx}(n; e; k)\ (\text{in } n_1), \quad \text{rep}_k(\alpha) \downarrow w\ (\text{in } n_2), \\ \text{getrecr}(\rho_\epsilon(m); b; f[e/h, \texttt{cpx}(n; \texttt{lcpx}(n; e; k-1))/t, \texttt{cpx}(n; \text{rep}_{k-1}(\alpha))/z]) \end{array}}{\texttt{listind}(l; b; h,t,z.f) : \alpha \downarrow w\ (\text{in } n_1 + n_2)}$$

Fig. 12. Symbolic list induction rule

The representative r of this term is $\texttt{cpx}(\rho(m); \lambda x.\texttt{cpx}(\rho_{\texttt{a}}(m, \texttt{time}(x)); \texttt{Ax}))$. Symbolic evaluation of (*hyper p q*) yields

$$1 + (1 + 0 + \texttt{time}(p) + 1 + \rho(p)) + \rho_{\texttt{a}}(p, \texttt{time}(q))$$

where $\rho(m) \equiv 0$ and

$$\rho_{\texttt{a}}(m, x) = \begin{cases} 2 + \rho(m-1) + \rho_{\texttt{a}}\big(m-1, \\ \quad 3 + \rho(m-1) + \rho_{\texttt{a}}(m-1, \texttt{time}(x))\big) & \text{if } m > 0 \\ \texttt{time}(x) & \text{if } m = 0 \end{cases} .$$

This reduces to

$$\rho_{\texttt{a}}(m, x) = \begin{cases} \rho_{\texttt{a}}(m-1, \rho_{\texttt{a}}(m-1, \texttt{time}(x)) + 3) + 2 & \text{if } m > 0 \\ \texttt{time}(x) & \text{if } m = 0 \end{cases}$$

or

$$\rho_{\texttt{a}}(m, x) = 5(2^m - 1) + \texttt{time}(x)$$

so that the complexity of (*hyper p q*) is $3 + \texttt{time}(p) + \texttt{time}(q) + 5(2^p - 1)$.

3.4 List Recursion

List recursion is very similar to primitive recursion; in fact, it uses the same terms for representative and recurrence variables and relies on the same procedure for infering recurrence equations (cf. Figure 12). As an example, the function

$$\textit{sumrest} == \lambda L.\texttt{listind}(L; 0; u, us, v.\texttt{listind}(us; 0; x, xs, y.x + y))$$

sums up all but the first elements. The complexity of (*sumrest* $\texttt{lcpx}(n; e; m)$) is $1 + \texttt{tmax}(1; 2m + mn + (m-1)\,\texttt{time}(e))$.

Returning to our example from Section 3.3, we now wrap a summation around our lazy list to enforce computation of all list elements. The complexity of

$$\texttt{listind}((\textit{table}\ \lambda x.\texttt{cpx}(n; \texttt{Ax}))\ m); 0; h, t, z.h + z)$$

becomes $5 + 5m + mn + \texttt{time}(m)$.

3.5 Soundness

To prove the soundness of the symbolic evaluation rules with respect to the profiling semantics in Figure 2 and time complexity definition (1), we have to show that

$$t_{\bar{v}} \downarrow^s w_{\bar{v}}\ (\text{in } n_{\bar{v}}) \quad \text{implies} \quad \forall \bar{u}.\text{type}(\bar{u}) = \text{type}(\bar{v}) \supset t_{\bar{u}} \downarrow w_{\bar{u}}\ (\text{in } n_{\bar{u}}),$$

where $\downarrow^s$ denotes symbolic evaluation and $\downarrow$ denotes regular evaluation. Let's assume that $t_{\bar{v}} \downarrow^s w_{\bar{v}}$ (in $n_{\bar{v}}$) for some symbolic $t_{\bar{v}}$ with free variables $\bar{v}$ and let $\bar{u}$ be any type-correct instantiation of $\bar{v}$. We prove $t_{\bar{u}} \downarrow w_{\bar{u}}$ (in $n_{\bar{u}}$) by induction on the derivation of $t_{\bar{v}} \downarrow^s w_{\bar{v}}$ (in $n_{\bar{v}}$):

(var) The base case $t_{\bar{v}} = v$ is trivial by our definition of v^* and $\mathtt{time}(v)$.

(arith) Let $k_i \downarrow^s k'_i$ (in n_i) for $i = 1, 2$ and $\mathtt{add}(k_1; k_2) \downarrow^s \mathtt{add}^*(k'_1; k'_2)$ (in $n_1 + n_2 + 1$). Then by inductive hypothesis, $\hat{k}_i \downarrow \hat{k}'_i$ (in $\hat{n}_i$) for any instantiation $\hat{k}_i, \hat{n}_i$ of k_i, n_i. Given our profiling semantics, we have $\mathtt{add}(\hat{k}_1; \hat{k}_2) \downarrow \hat{k}'_1 + \hat{k}'_2$ (in $\hat{n}_1 + \hat{n}_2 + 1$), where by definition $\hat{k}'_1 + \hat{k}'_2$ instantiates $\mathtt{add}^*(k'_1; k'_2)$ and $\hat{n}_1 + \hat{n}_2 + 1$ instantiates $n_1 + n_2 + 1$.

(comp) Let $k_i \downarrow^s k'_i$ (in n_i) for $i = 1, 2$ and $\mathtt{indet}(u; v) \downarrow^s w$ (in n_3). Then again by inductive hypothesis, $\hat{k}_i \downarrow \hat{k}'_i$ (in $\hat{n}_i$) for any instantiation $\hat{k}_i, \hat{n}_i$ of k_i, n_i. Now if $\hat{k}_1 < \hat{k}_2$ and $\hat{u} \downarrow \hat{w}$ (in $\hat{n}_3$), then $\mathtt{less}(\hat{k}_1; \hat{k}_2; \hat{u}; \hat{v}) \downarrow \hat{w}$ (in $\hat{n}_1 + \hat{n}_2 + \hat{n}_3 + 1$) for some instantiations $\hat{u}$, $\hat{v}$, $\hat{w}$. Since $\hat{u}$ is also an instantiation of $\mathtt{indet}(u; v)$, we have by inductive hypothesis that $\hat{w}$ and $\hat{n}_3$ are instantiations of w and n_3, respectively, which proves our claim. The case $\hat{k}_1 > \hat{k}_2$ is analogous.

(indet) Let $p \downarrow^s \mathtt{indet}(p_1; p_2)$ (in n) and $\mathtt{op}(p_i; \bar{u}) \downarrow^s w_i$ (in n_i) for $i = 1, 2$. Then by inductive hypothesis, $\hat{p} \downarrow \hat{p}'$ (in $\hat{n}$) for any instantiations $\hat{p}, \hat{p}', \hat{n}$ of p, p_1, n. Now let $\hat{u}, \hat{w}_i, \hat{n}_i$ be instantiations of $\bar{u}, w_i, n_i$. By inductive hypothesis, $\mathtt{op}(\hat{p}'; \hat{u}) \downarrow \hat{w}_1$ (in $\hat{n}_1$). Clearly, $\hat{n}_1 \leq \max(\hat{n}_1, \hat{n}_2)$ and thus $\mathtt{op}(\hat{p}; \hat{u}) \downarrow \hat{w}_1$ (in $\hat{n} + \mathtt{tmax}(\hat{n}_1; \hat{n}_2)$), where $\hat{w}_1$ instantiates $\mathtt{indet}(w_1; w_2)$. The case where $\hat{p}'$ instantiates p_2 is analogous.

(ind), (listind) By our construction, an inductive term $t == \mathtt{ind}(k; b; i,z.f) : \alpha$ and its representative $r == \mathrm{rep}_m(\alpha)[k^*/m]$ have the same computational behavior. In other words, in any permissible context C (i.e. excluding those where the `Ax` terms of the representative would enter the computation path), we have $C[t] \downarrow^s w$ (in n) iff. $C[r] \downarrow^s w'$ (in n). Thus, the soundness of (ind) is reduced to the soundness of (indbase) and (indstep), which are covered below. The case for `listind` is analogous.

(other) All other symbolic rules are identical to their corresponding standard rule so that their soundness proof is trivial.

3.6 Symbolic Call-by-Value Reduction

In addition to implementing the profiling semantics of Figure 5 for symbolic evaluation, we need to adopt the initial step of the recurrence generation in Figure 10. Instead of matching the representative $r == \mathrm{rep}_{m-1}(\alpha)$ with $f[m/i, r/z]$, we now use the *flat representative* $r' == r[0/\rho_d]$ to match r with $f[m/i, r'/z]$. To account for the cost of unfolding the inductive term, we add $\rho_d(m-1)$ to the result n_m in each rule of Figure 10.

4 Recurrence Solving

A central part of the ACAp system is devoted to symbolic manipulation of algebraic expressions. Alas, NUPRL is not well-equipped for this task: its single rewriting function `arith_simplify_term` performs certain undesirable transformations to

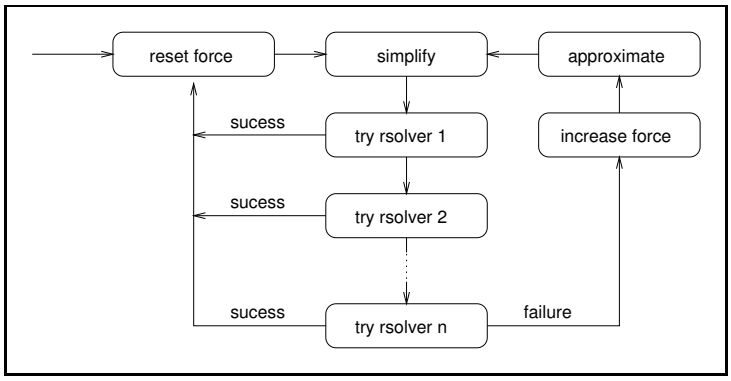

Fig. 13. The recurrence solver

our new terms `power`, `frac`, and `tmax`. We thus implemented a new simplification routine that also knows about some trivial identities like $\texttt{tmax}(a;a) = a$ or $1^a = 1$.

In addition to term simplification, solving non-trivial recurrence relations requires some means to support summation $\sum_{i=p}^{q} \varphi_i$ and function iteration $f^p(x)$ for symbolic p and q. Two rudimentary *arithmetical support functions* `arith_sum` and `arith_iterate` return closed expressions for sums over c, cq^i, and $ci^k (k \leq 3)$ terms and for iterations of linear functions f.

These simple routines handle the most frequent, trivial cases fast and efficiently, but they are inadequate for any "smart" reasoning. For more advanced tasks, we thus harness the computational power of the MATHEMATICA symbolic algebra system (Wolfram, 1996) for both term rewriting and recurrence solving (cf. Section 4.3).

4.1 The Recurrence Solver

The recurrence solver is designed as an extensible collection of individual *rsolver modules* that are tailored to obtain solutions for particular recurrence classes identified by pattern matching. Each rsolver takes a recurrence equation $(\rho_d(m, \bar{x}); \varphi(\bar{x}); \psi(m, \bar{x}))$ and returns a (possibly empty) list of substitutions (ρ_d, χ) such that $\varphi(\bar{x}) = \chi(0, \bar{x})$ and $\psi(m, \bar{x}) = \chi(m, \bar{x})$ hold. The main routine repeatedly sweeps the rsolver modules across the list of unsolved recurrences; computationally inexpensive rsolvers are tried first before more costly modules are applied as well. Since inductive terms might generate large recurrence systems involving many variables, some modules need to examine multiple recurrences at a time. In practice, however, many systems are of triangular shape and can easily be solved by Gaussian elimination using single-equation rsolvers.

The current prototype features rsolvers for linear equations with constant coefficients and various linear equations involving `tmax` terms. We distinguish between *internal* modules written in ML and *external* modules implemented in MATHEMATICA's functional programming language. The internal rsolver modules solve most of the standard cases, which in our experience account for 90% or more of all recurrence equations generated.

4.1.1 Linear Recurrences with Constant Coefficients

The built-in `const_linear` module is the bread-and-butter solver of the system and applies to very simple linear recurrence equations with constant coefficients:

$$\rho(m,\bar{x}) = \begin{cases} c_1\rho(m-1,p(\bar{x})) + c_0(\bar{x}) & \text{if } m > 0 \\ \rho_0(\bar{x}) & \text{if } m = 0 \end{cases}$$

As indicated by our notation, c_0 may not depend on m and c_1 may not depend on m or $\bar{x}$. The main solver routine builds a dependency graph so that indirect references of dependent variables can be identified. For example, `const_linear` applies to ρ_1 in

$$\begin{aligned} \rho_1(m,y) &= \rho_2(z)\,\rho_1(m-1,y+1) \\ \rho_2(n) &= \rho_2(n-1)+1, \end{aligned}$$

but not in

$$\begin{aligned} \rho_1(m,y) &= \rho_2(z)\,\rho_1(m-1,y+1) \\ \rho_2(n) &= \rho_2(m-1)+1, \end{aligned}$$

so that in the latter case ρ_2 need to be determined first. The solution to the general equation is

$$\rho(m,\bar{x}) = c_1^m \cdot \rho_0(p^m(\bar{x})) + \sum_{j=0}^{m-1} c_1^j c_0(p^j(\bar{x})).$$

The current arithmetical support functions restrict the applicability of `const_linear` to linear functions $p(\bar{x}) = \bar{a}_1 \cdot \bar{x} + \bar{a}_0$, where $\bar{a}_1$ is a vector of integers, $\bar{a}_0$ is a vector of terms not containing $\bar{x}$, and '$\cdot$' denotes component-wise multiplication.

The `unit_linear_polym` module is very similar to `const_lin` in that it applies to linear recurrence equations, but it allows for an additional polynomial while requiring a unit coefficient:

$$\rho(m,\bar{x}) = \begin{cases} \rho(m-1,p(\bar{x})) + b_0(\bar{x}) + \sum_{i=1}^{k} b_i m^i & \text{if } m > 0 \\ \rho_0(\bar{x}) & \text{if } m = 0 \end{cases}$$

The solution to this recurrence is

$$\rho(m,\bar{x}) = \rho_0(p^m(\bar{x})) + \sum_{j=0}^{m-1} b_0(p^j(\bar{x})) + \sum_{i=1}^{k}\sum_{j=0}^{m} b_i j^i.$$

The support functions further require that p be a linear function and coefficients $b_1,\ldots,b_k$ be integer constants with k no larger than 3.

4.1.2 Linear Recurrences Involving `tmax` terms

The `linear_tmax` resolver applies to linear recurrences in which the recurrence variable is part of a `tmax` term:

$$\rho(m,\bar{x}) = \begin{cases} \texttt{tmax}(\rho(m-1,p(x)) + c_0(x); a_1 m + a_0(x)) & \text{if } m > 0 \\ \rho_0(\bar{x}) & \text{if } m = 0 \end{cases}$$

Force	Approximation
0	none
1	$\texttt{tmax}(\sum_{j=1}^{n} a_j u_j + a_0; \sum_{j=1}^{n} b_j u_j + b_0) \mapsto \sum_j^n \texttt{tmax}(a_j; b_j) u_j + \texttt{tmax}(a_0, b_0)$ where $a_j, b_j > 0$ for $j = 1, \ldots, n$
2	$\texttt{tmax}(\sum_{j=1}^{n} a_j u_j + a_0; \sum_{j=1}^{n} b_j u_j + b_0) \mapsto \sum_j^n \texttt{tmax}(a_j; b_j) u_j + \texttt{tmax}(a_0, b_0)$
3	$\texttt{tmax}(u; v) \mapsto u + v$

Fig. 14. `tmax` approximations

A solution to this recurrence equation is

$$\rho(m, \bar{x}) = \max\big(\rho_0(p^m(\bar{x})) + \sum_{j=0}^{m-1} c_0(p^j(\bar{x})), \max_{k=1,\ldots,m} a_1\, k + a_0(p^{m-k}(\bar{x})) + \sum_{j=0}^{m-k-1} c_0(p^j(\bar{x}))\big).$$

Again, the arithmetical support functions cannot generate closed expressions in this generality. For the special case $a_0(\bar{x}) = a_0$, $c_0(\bar{x}) = c_0$, however, above solution simplifies to

$$\rho(m, \bar{x}) = \max\big(\rho_0(p^m(\bar{x})) + mc_0, \max_{k=1,\ldots,m} k\, a_1 + a_0 + (m-k)c_0\big),$$

which may be abbreviated further to

$$\rho(m, \bar{x}) \leq \max(a_1, c_0)\, m + \max(\rho_0(p^m(\bar{x})), a_0).$$

This last bound is implemented in the system.

4.2 Approximation Heuristics

Because of indeterminism, `tmax` terms are ubiquitous in raw complexity expressions returned by the symbolic evaluator. Many of these terms are trivial as they originate from intermediate reduction steps that do not depend on some previous evaluation of an indeterminate term. More complicated expressions, however, might contain other `tmax` terms or recurrence variables as subterms.

Recurrences involving `tmax` terms are considerably harder to solve than regular ones. To improve the performance of the recurrence solver, a collection of `tmax` *approximations* heuristically replace `tmax` terms by less complex terms at the cost of looser bounds. Since `tmax` terms are so common in complexity expression, these heuristics have a profound influence on the quality of the bounds obtained.

Recurrence solving is an iterative process. First, the system applies all known simplifications as described in Section 4. After this step, the recurrence solver sweeps the individual rsolver modules over the simplified equations, with additional simplification after a solution is found and substituted. This process is repeated until no further recurrences can be solved. At this point, the system applies some `tmax` approximation before it resumes the rsolver cycle (cf. Figure 13).

Approximations have to be controlled carefully, or they will spoil the analysis.

Obviously, more accurate approximations should be tried first before reverting to more crude substitutions. The ACAp system classifies approximations into several *force levels* of increasing inaccuracy. If gentle approximations of an expression do not result in solving any new recurrences, the system has to apply more force while incurring looser bounds. The force levels currently implemented in the system are shown in Figure 14. Note that levels 1 and 2 differ only in their restriction on the coefficients of the polynomials involved. This discrimination was necessary because the level 2 approximation might result in very crude bounds if one of the polynomial terms u_j involves recurrence variables.

4.3 Solving Recurrences With Mathematica

As we mentioned earlier, the recurrence solver employs the MATHEMATICA computer algebra system (Wolfram, 1996) to solve more complicated difference equations generated by the symbolic evaluator. MATHEMATICA's built-in recurrence solver `RSolve`, however, is not adequate for our needs: Like similar packages of other systems such as MAPLE or MUPAD, it relies on power series expansion and thus does not support functions of several variables.

In our setting, a general linear recurrence equation with constant coefficients has the form

$$r(m,x,\ldots,z) = \begin{cases} \sum_i a_i\, r(m-1,x+b_{xi},\ldots,z+b_{zi}) & \\ \qquad +q(m,x,\ldots,z) & \text{for } m>0\ , \\ p(x,\ldots,z) & \text{for } m=0 \end{cases} \tag{2}$$

where p and q are arbitrary polynomials. Following the ideas in (Levy & Lessman, 1961), we define operators Ξ and Δ such that

$$\begin{aligned} \Xi_\nu\varphi &:= \varphi[\nu+1/\nu] \\ \Delta_\nu\varphi &:= \varphi[\nu+1/\nu]-\varphi \end{aligned}$$

for any variable ν. So for example, $\Xi_m r(m,x) = r(m+1,x)$ and $\Delta_x(x^2-y^2) = (x+1)^2-x^2$. We might think of Δ_ν as being the discrete analog of the differential operator $\partial/\partial\nu$.

Operators Ξ and Δ are obviously distributive and commutative and satisfy the *index law* $\Xi^p\Xi^q = \Xi^{p+q}$, $\Delta^p\Delta^q = \Delta^{p+q}$. Hence, we might subject Ξ and Δ to common algebraic manipulations with the usual identities like $(\Delta+1)^2 = \Delta^2+2\Delta+1$ or $e^\Xi = 1+\Xi+\Xi^2/2!+\Xi^3/3!+\cdots$ (see (Levy & Lessman, 1961) for details). In particular, we have the fundamental relationship

$$\Xi_\nu = \Delta_\nu + 1,$$

where 1 is the identity operator.

To solve a particular instance of (2), we first rewrite the step case $m>0$ by substituting $m+1$ for m and using the Ξ operator:

$$\Xi_m r(m,x,\ldots,z) = \sum_i \left(a_i \prod_\nu \Xi_\nu^{b_{\nu i}} \right) r(m,x,\ldots,z) + q(m+1,x,\ldots,z). \tag{3}$$

To find a *particular solution* $r_p(m, x, \ldots, z)$ that satisfies (3) but not necessarily (2) for $m = 0$, we substitute $\Delta + 1$ for Ξ and move all operators to the left-hand side of the equation, yielding

$$\Phi(\bar{\Delta}) r_p(m, x, \ldots, z) = q(m+1, x, \ldots, z) \tag{4}$$

for some polynomial $\Phi(\bar{\Delta}) = \Phi(\Delta_m, \Delta_x, \ldots, \Delta_z)$. Now we can formally solve for r_p, arriving at

$$r_p(m, x, \ldots, z) = \Phi(\bar{\Delta})^{-1} q(m+1, x, \ldots, z)$$

or, by pulling the constant factor α out of Φ,

$$\begin{aligned} r_p(m, x, \ldots, z) &= 1/\alpha \; (1 - (1 - \Phi(\bar{\Delta})/\alpha))^{-1} q(m+1, x, \ldots, z) \\ &=: 1/\alpha \; (1 - \varphi)^{-1} q(m+1, x, \ldots, z). \end{aligned}$$

Expanding $1/(1-\varphi)$ to its formal power series $1 + \varphi + \varphi^2 + \varphi^3 + \cdots$, we continue

$$\begin{aligned} r_p(m, x, \ldots, z) &= 1/\alpha \; (1 + \varphi + \varphi^2 + \cdots) \, q(m+1, x, \ldots, z) \\ &= 1/\alpha \; (1 + \varphi + \varphi^2 + \cdots + \varphi^k) \, q(m+1, x, \ldots, z) \end{aligned}$$

for some finite k, since φ does not have a constant term and q is a finite polynomial so that $\varphi^j q = 0$ for all sufficiently large j. Hence, we have a finite procedure to compute a particular solution r_p for (3).

As a simple example, let

$$r(m, x, y) = \begin{cases} a\, r(m-1, x+1, y) + r(m-1, x, y+2) + mx + y & \text{for } m > 0 \\ xy & \text{for } m = 0 \end{cases}.$$

Since the argument y is increased by 2, we redefine $\Xi_y \varphi$ to be $\varphi[\nu + 2/\nu]$ to simplify our presentation. The step case is then equivalent to $\Xi_m r = (a\Xi_x + \Xi_y) r + mx + x + y$ or

$$(\Delta_m - a\Delta_x - \Delta_y - a) r = mx + x + y,$$

so $\alpha = -a$ and $\varphi = -\Delta_m/a + \Delta_x + \Delta_y/a$. Expanding Φ up to φ^2, simplification with MATHEMATICA yields

$$r_p(m, x, y) = \frac{1}{-a^2} \left(x + a((m+1)(x-1) + y) - 4 \right)$$

as the particular solution for $m > 0$.

To find a solution that satisfies (2) for $m = 0$, we first determine a solution r_h to the *homogeneous* recurrence equation

$$\Xi_m r(m, x, \ldots, z) = \sum_i \left(a_i \prod_\nu \Xi_\nu^{b_{\nu i}} \right) r(m, x, \ldots, z). \tag{5}$$

As in the theory of differential equations, all solutions to (2) for $m > 0$ can be expressed as a linear combination $r_p + \alpha\, r_h$ of the homogeneous and the particular solution. The case $m = 0$ then becomes a *boundary condition* that has to be satisfied by determining an appropriate value for α.

The operators on the right-hand side of (5) do not interfere with m, so we may unfold above relation to get

$$\begin{aligned} r_h(m,x,\ldots,z) &= \Gamma\, r_h(m-1,x,\ldots,z) \\ &= \Gamma^2\, r_h(m-2,x,\ldots,z) \\ &= \cdots \\ &= \Gamma^m\, r_h(0,x,\ldots,z) \\ &= \Gamma^m\, A(x,\ldots,z) \end{aligned}$$

where $\Gamma = \sum(a_i \prod \Xi_\nu^{b_{\nu i}})$. The function A is arbitrary, so by picking $A(x,\ldots,z) = p(x,\ldots,z) - r_p(0,x,\ldots,z)$, we ensure that the boundary condition is satisfied for $\alpha = 1$. To eliminate the operators, we repeatedly use binomial expansion of $\Gamma = \gamma_1 + \gamma_2 + \cdots + \gamma_k$ and apply any operators in $\gamma_i = \Delta_{\nu_1}\Delta_{\nu_2}\cdots\Delta_{\nu_m}$:

$$\begin{aligned} r_h(m,x,\ldots,z) &= (\gamma_1 + \Gamma_1)^m A_0(x,\ldots,z) \\ &= \sum_{i=0}^{m} \binom{m}{i} \gamma_1^i (\Gamma_1^{m-i} A_0(x,\ldots,z)) \\ &= \cdots \\ &= \sum_{i=0}^{m} \binom{m}{i} \gamma_1^i A_{m-1}(x,\ldots,z) \\ &= \sum_{i=0}^{m} \binom{m}{i} A_m(x,\ldots,z) \end{aligned}$$

Since $A_0 = A$ is a given polynomial, the remaining polynomials $A_j = \gamma_{m-j}^k A_{j-1}$ can be computed explicitly. The only difficulty now is to eliminate the sum from the last line, since A_m may involve the variable m. MATHEMATICA seems quite powerful in obtaining a closed expression, but we know of several cases involving four or five symbolic constants where the system exhausted the available main memory of 2 GB when attempting to find a solution.

To complete the example from above, we have

$$\begin{aligned} r_h(m,x,y) &= \sum_{i=0}^{m} \binom{m}{i} (a\Xi_x)^i (\Xi_y)^{m-i} (xy - r_p(m,x,y)) \\ &= \sum_{i=0}^{m} \binom{m}{i} (a\Xi_x)^i (x(y+2m-2i) - r_p(m,x,y+2m-2i)) \\ &= \sum_{i=0}^{m} \binom{m}{i} a^i (x+i)(y+2m-2i) - a^i r_p(m,x+i,y+2m-2i), \end{aligned}$$

which MATHEMATICA simplifies to

$$\begin{aligned} r(m,x,y) &= r_p(m,x,y) + r_h(m,x,y) \\ &= \frac{1}{a^2}\Big((a+1)^{m-2}\big((x-4) + a(3m+3x+y-9) \\ &\qquad + a^2(2m(2+x)+3x+2y+xy-6) \\ &\qquad + a^3(2m^2+m(2x+y-1)+x+y+2xy-1) \\ &\qquad + a^4(m+x)y\big) - a(m(x-1)+x+y-1) - x + 4\Big). \end{aligned}$$

5 Examples

This section illustrates the combined use of NUPRL and the ACA*p* system for program extraction and complexity analysis. In the first example, we synthesize an efficient algorithm for the Maximum Segment Sum problem and determine an upper bound on its worst-case time complexity. Due to space limitations, the formal proof of the problem specification had to be omitted; the interested reader can find it in (Benzinger, 1999). The second analysis shows how call-by-name evaluation allows efficient computation of the minimum of an integer list by sorting it. We conclude the section by contrasting the different run-time behavior of call-by-name and call-by-value reduction.

5.1 Maximum Segment Sum

Let a be an integer list $a_1 :: a_2 :: \cdots :: a_n$ of length n. A *segment* of a is any consecutive subsequence $a_i :: a_{i+1} :: \cdots :: a_j$, $1 \leq i, j \leq n$, where the segment is empty if $i > j$. A *prefix* is a segment where $i = 1$.

We use NUPRL to synthesize an efficient solution to the well-known *Maximum Segment Sum* problem originally presented by Jon Bently and David Gries (Gries, 1982): Given an integer list a, find the largest element sum M of all segments of a, i.e.

$$M = \max_{1 \leq i,j \leq n} \sum_{k=i}^{j} a_k.$$

The maximum segment sum is a popular textbook example for a problem that has a relatively simple specification, but whose most obvious solutions are computationally inefficient. The straightforward imperative approach shown in Figure 15 has running time $O(n^2)$, which is worse than the linear algorithm by Gries given in Figure 16. Subsequently, Bates and Constable formally derived an efficient solution in a functional programming language with the PRL system (Bates & Constable, 1985). Although their proof yields the functional equivalent of the linear imperative solution, they did not have the means to formally assess the actual complexity of their program.

In our new proof, two abstractions $\texttt{pfxsum}(L;k)$ and $\texttt{segsum}(L;j;k)$ define the sum of prefix $L_1 :: \cdots :: L_k$ and segment $L_k :: \cdots :: L_{j+k-1}$, respectively. The abstractions are supplemented by definitions for display forms and well-formedness

```
m := 0
for i := 1 to n do
   s := 0
   for j := i to n do
      s := s + a[j]
      m := max(m, s)
```

Fig. 15. The naïve maxsegsum algorithm

```
l, m := 0
for i := 1 to n do
   l := max(l + a[i], 0)
   m := max(m, l)
```

Fig. 16. Gries' linear algorithm

proofs as well as some simple rewrite tactics to automate recurrent manipulations of these terms.

The proof mainly follows the lines of (Bates & Constable, 1985), but the reader should note that we consider the empty segment a valid solution. To enforce the desired computational behavior, the basic maxsegsum specification is strengthened by asserting that for every list L, there exists a maximum prefix sum *pfxL* and a maximum segment sum *maxL*:

$$\forall L\colon \mathbb{Z}\ \texttt{list}.\ \big((\exists \mathit{pfxL}\colon \mathbb{N}.\ (\forall l\colon \mathbb{N}.\ l \le \|L\| \supset \mathit{pfxsum}(L;l) \le \mathit{pfxL})) \ \&\\ (\exists \mathit{maxL}\colon \mathbb{N}.\ (\forall j,k\colon \mathbb{N}.\ j \le \|L\| \supset j+k \le \|L\| \supset \mathit{segsum}(L;j;k) \le \mathit{maxL}))\big)$$

The proof proceeds by list induction on L. The base case $L = \texttt{[]}$ is trivial, since the maximum segment sum of the empty list is 0. For the step case $L = u :: v$, we first show the existence of the maximum prefix sum of L. By induction hypothesis, *pfxL* is the maximum prefix sum of v. Hence, $u + \mathit{pfxL}$ is the maximum prefix sum of $u :: v$ if $u + \mathit{pfxL} > 0$; otherwise, `[]` is the maximal prefix of $u :: v$. Using *pfxL*, we can now prove the existence of the maximum segment sum of L. By induction hypothesis, *maxL* is the maximum segment sum of v. Any maximal segment of L that is not also maximal segment of v must clearly begin with u. Since *pfxL* is the maximal sum of all these segments, $\max(\mathit{pfxL}, \mathit{maxL})$ is the maximum segment sum of $u :: v$, q.e.d.

The resulting extract is given in Figure 17. As usual, dead code, introduced as proof fragments by existential quantification and arithmetical reasoning, obfuscates the actual algorithm. Analysis with the ACAp system yields

```
M> let mss = f_of_thm ‘maxsegsum‘ ;;
M> fcc mss ⌜lcpx(0;0;m)⌝ ;;
5 + 17 * m * 1/2 + 3 * m * m * 1/2 : term
```

as an upper bound on the time complexity of the algorithm. The suggested quadratic running time is surprising at first but becomes understandable upon closer examina-

```
λL.let <pfxL,%1> = rec-case(L)
    of [] =>
       <0, λl,%,.Ax, 0, λj,k,%,%1,.Ax>
     | u::v =>
       %.let <pfxL,%1> = % in
         let <%2,%3> = %1 in
         let <maxL,%4> = %3 in
         case if 0<(u + pfxL) then inr (λ,.Ax)  else (inl (λ.Ax) )
         of inl(%) =>
               <0,
                λl,%5.case l ≤z 0 of inl(x) => λ.Ax | inr(y) => λ.Ax,
                maxL,
                λj,k,%5,%6.case j ≤z 0
                   of inl(x) => case k ≤z 0
                                   of inl(x) => λ.Ax | inr(y) => λ.Ax
                    | inr(y) => λ.Ax>
          | inr(%6) =>
               <u + pfxL,
                λl,%.case l ≤z 0 of inl(x) => λ.Ax | inr(y) => λ.Ax,
                case if maxL<(u + pfxL)
                     then inr (λ,.Ax)  else (inl (λ.Ax) )
                of inl(%) =>
                      <maxL,
                       λj,k,%7,%8.case j ≤z 0
                       of inl(x) =>
                             case k ≤z 0
                             of inl(x) => λ.Ax | inr(y) => λ.Ax
                        | inr(y) => λ.Ax>
                 | inr(%8) =>
                      <u + pfxL,
                       λj,k,%,%9.case j ≤z 0
                       of inl(x) =>
                             case k ≤z 0
                             of inl(x) => λ.Ax | inr(y) => λ.Ax
                        | inr(y) => λ.Ax>
                >
   in
   let <%2,%3> = %1 in %3
```

Fig. 17. Maximum segment sum extract

tion of the algorithm: in each iteration, the term $u + pfxL$ might be reduced multiple times. Counting the number of reduction steps for some concrete worst-case lists of length m indeed yields

m:	0	1	2	3	4	5	6	7	8	9	...
$\rho(m)$:	4	14	27	43	62	84	109	137	168	202	...

or $\rho(m) = 4 + 17m/2 + 3m^2/2$.

Unwinding the program with its nested pairs of intermediate values that result from different iterations is fairly involved. If we look at the worst case of recomput-

$$
\begin{aligned}
\rho(l) &= 7 + \rho(l-1) + \rho_1(l-1) + \rho_{22}(l-1) \\
\rho_1(l) &= 1 + \rho_1(l-1) \\
\rho_{21\mathtt{aa}}(l, l', x) &= 9 + \mathtt{time}(l') \\
\rho_{22}(l) &= 3 + \rho_1(l-1) + \rho_{221}(l-1) \\
\rho_{221}(l) &= \mathtt{tmax}_2(\mathtt{tmax}_5(1 + \rho_1(l-1); \rho_{221}(l-1)); \rho_{221}(l-1)) \\
\rho_{222\mathtt{aaaa}}(l, j, k, x, y) &= 8 + \mathtt{time}(j) + \mathtt{tmax}_6(1; 10 + \mathtt{time}(k))
\end{aligned}
$$

Fig. 18. Non-trivial recurrence equations for maxsegsum

ing $maxL$ in each iteration, we arrive at

$$T(n) = 2\,T(n-1) + c$$

as a crude estimate of our real time complexity. This recurrence has an exponential solution, which is a far worse than our computer-generated bound. In fact, when we wrote an uncluttered version of the algorithm by hand, the resulting program exhibited exponential running time both in experiments and when analyzed by ACAp. The maximum segment sum example thus rather convincingly shows the usefulness of the ACAp system: as programs grow larger, manual complexity analysis quickly becomes infeasible.

In order to determine the bounds for the maximum segment sum function, the system generates 15 recurrence equations, of which all but 6 are trivial and eliminated during the first round of recurrence solving. The remaining simplified equations, shown in Figure 18, are solved by consecutive sweeps of the internal rsolvers at different force levels.

5.2 Insertion Sort and Maximum

A popular example to contrast call-by-name and call-by-value evaluation is computing the minimum of an integer list by sorting the list and returning the head of the sorted list. If we use insertion sort, sorting a list of length m will take $O(m^2)$ operations. Extracting the head of a list takes constant time, so the composition of these two operations using call-by-value takes $O(m^2)$ steps. With call-by-name evaluation, however, only the head of the sorted list has to be computed, and this is done in $O(m)$ steps. Hence, using call-by-name, the minimum can be determined in linear time.

To make the algorithms more readable, we encoded them by hand rather than synthesizing extracts with NUPRL. Given the abstractions

$$
\begin{aligned}
\mathtt{ielem}(x; L) &== \mathtt{listind}(L;\ x :: \mathtt{[]};\ h,t,z.\mathtt{ifless}(x;\ z;\ x :: h :: t;\ h :: z)) \\
\mathtt{isort}(L) &== \mathtt{listind}(L;\ \mathtt{[]};\ h,t,z.\mathtt{ielem}(h; z)),
\end{aligned}
$$

sorting a list of length m takes $O(m^2)$ time if we browse through the whole list:

```
M> cc ⌜listind(isort(lcpx(k;0;m)); 0; h,t,z.h+z)⌝ ;;
3 + 10 * m + k * m + 6 * m * m : term
```

Using lazy evaluation, however, we can take the head of the sorted list to compute the minimum in $O(m)$ time:

```
M> cc ⌜hd(isort(lcpx(k;0;m)))⌝ ;;
4 + 8 * m + k * m : term
```

5.3 Call-by-Name versus Call-by-Value

As we have seen in the previous examples, call-by-name evaluation can be a boon but also a curse. For NUPRL extracts, call-by-name reduction conveniently ignores dead code in the form of proof fragments. From a practitioner's point of view, however, call-by-value if often the more natural reduction strategy. If we repeat above analyses using the call-by-value semantics from Section 2.3, the Maximum Segment Sum becomes linear, as expected:

```
M> fccv mss ⌜lcpx(0;0;m)⌝ ;;
4 + 11 * m : term
```

On the other hand, computing the minimum of an integer list via insertion sort is now no longer efficient, as the complete sorted list will be computed, even though we need only the first element:

```
M> ccv ⌜hd(isort(lcpx(k;0;m)))⌝ ;;
4 + k * m + m * 1/2 + 7 * m * m * 1/2 : term
```

As more and better evaluation strategies for NUPRL become available, the ACAp system should be flexible enough to accommodate them.

6 Conclusion

We developed a cost model of an abstract call-by-name interpreter for NUPRL's standard term language and showed how alternative reduction strategies such as call-by-value are realized. The central part of the calculus are the `cpx` and `lcpx` terms, which allow the definition of abstract functions $\lambda x.\texttt{cpx}(n_x; t_x)$ and abstract lists $\texttt{lcpx}(n; e; k)$. The framework supports primitive recursive programs with first-order functions and lazy lists as well as higher-order functions whose inputs can be defined by abstract functions. Symbolic evaluation is introduced as a sound extension of standard evaluation to open terms. For automatic analysis, symbolic inductive terms are replaced by their representative, a non-recursive decomposition into first-order terms. The individual time complexities of these terms are translated into a system of multi-variable difference equations, which are solved by common techniques.

We designed and implemented a working prototype system (ACAp) to explore the potential and the pitfalls of our theoretical framework. The system automatically derives an upper bound on the time complexity of a given NUPRL extract or program with respect to either call-by-name or call-by-value semantics. Difference equations are solved by various modules written in NUPRL ML or MATHEMATICA's functional programming language.

6.1 Future Work

Some of our near term goals include improving the recurrence solver by writing additional MATHEMATICA packages for linear equations with non-constant coefficients and classes of non-linear equations, and by fine-tuning the `tmax` heuristics to improve the bounds for nested conditionals. Another plan to increase the usability of the system is to enhance the user dialog with the system to permit the interactive analysis of particularly involved programs.

Although the integer variable assumption works fairly well in practice, further work remains to be done. Some tactics, like complete induction, often require that functions be passed as an argument to the inductive term. Another example in which the integer variable assumption is violated involves induction on pairs consisting of the result and some computationally irrelevant proof fragment.

If we use the representative not only for inductive terms but also for free variables, we can discard the integer variable assumption completely. The only problem with this approach is that inductive terms of higher type, e.g.

$$\mathtt{ind}(k; \lambda f.f0; i,z.\lambda f.z(\lambda x.x+1))$$

will generate "higher order" difference equations, which we may write as

$$\begin{aligned}\rho_{\mathtt{a}}(m, f, f_{\mathtt{a}}) \quad = \quad & \rho_{\mathtt{a}}(m-1, f(m-1), 1 + f_{\mathtt{a}}(m-1, x)) \\ & + \rho_{\mathtt{a}}(m-1, 0, 1 + f(m-1) + f_{\mathtt{a}}(m-1, x+1)) + c\end{aligned}$$

Current work focuses on extending the present operator approach to these higher order equations.

Finally, we plan to abolish the current restriction to primitive recursive functions. Future versions of the ACAp system will be able to derive recurrence equations from general recursive abstractions

$$\mathtt{op}(x) == \phi(\mathtt{op}(\psi(x)))$$

involving the `Y` combinator, allowing more natural extracts in the style of conventional ML programs.

Acknowledgments

The author would like to thank Jason J. Hickey for his suggestions on clarifying several aspects of this paper and his inexhaustible patience in explaining the nooks and crannies of the NUPRL system. I am also much indebted to my adviser Robert L. Constable for his help in identifying the area and his gentle guidance during my work on this document. Finally, I would like to thank the referees for their feedback and suggested improvements.

References

Bates, Joseph L., & Constable, Robert L. (1985). Proofs as programs. *ACM transactions on programming languages and systems*, **7**(1).

Benzinger, Ralph. (1999). *Automated complexity analysis of Nuprl extracts.* M.Phil. thesis, Cornell University.

Bjerner, Bror, & Holmström, Sören. (1989). A compositional approach to time analysis of first order lazy functional programs. *4th international coneference on functional programming languages and computer architecture.*

Bjørner, Dines, & Jones, Cliff B. (1978). *The Vienna Development Method.* Lecture Notes in Computer Science, vol. 61. Springer.

Caldwell, James L. (1997). Moving Proofs-as-Programs into practice. *IEEE international conference on automated software engineering.*

Constable, Robert L., *et al.* . (1986). *Implementing mathematics with the Nuprl Proof Development System.* Prentice-Hall.

Constable, Robert L., Jackson, Paul B., Naumov, Pavel, & Uribe, Juan. (1998). Constructively formalizing automata. *Proof, language and interaction: Essays in honour of Robin Milner.* MIT Press, Cambridge.

Coquand, Thierry, & Huet, Gérard. (1985). Constructions: A higher order proof system for mechanizing mathematics. *Eurocal '85.* Lecture Notes in Computer Science, vol. 203. Springer.

Flajolet, Philippe, Salvy, Bruno, & Zimmermann, Paul. (1990). *Automatic average-case analysis of algorithms.* Tech. rept. 1233. INRIA.

Gordon, M. J. C., & Melham, Thomas F. (1993). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press.

Greiner, John. (1997). *Semantics-based parallel cost models and their use in provably efficient implementations.* Ph.D. thesis, Carnegie-Mellon University.

Gries, David. (1982). *A note on the standard strategy for developing loop invariants and loops.* Tech. rept. TR 82-531. Cornell University.

Hafızogulları, Ozan, & Kreitz, Christoph. (1998). *Dead code elimination through type inference.* Tech. rept. Cornell University.

Hickey, Jason J. (2000). *MetaPRL.* Ph.D. thesis, Cornell University.

Hickey, Timothy, & Cohen, Jacques. (1988). Automating program analysis. *Journal of the ACM*, **35**(1), 185–220.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, **12**(10), 576–580.

Lawall, Julia L., & Mairson, Harry G. (1996). Optimality and efficiency: What isn't a cost model of the lambda calculus? *Pages 92–101 of: ACM international conference on functional programming.*

Levy, H., & Lessman, F. (1961). *Finite difference equations.* Macmillan.

Martin-Löf, Per. (1979). An intuitionistic theory of types: Predicative part. *Logic colloquium.*

Métayer, Daniel Le. (1988). ACE: An automatic complexity evaluator. *Transactions on programming languages and systems*, **10**(2).

Nogin, Alexsey. (1997). *Improving the efficiency of Nuprl proofs.* Tech. rept. TR97-1643. Cornell University.

Owre, S., Rushby, J. M., & Shankar, N. (1992). PVS: A prototype verification system. *Lecture notes in computer science*, **607**.

Paulin-Mohring, Christine. (1989). *Extraction in the calculus of constructions.* Ph.D. thesis, University of Paris VII.

Paulin-Mohring, Christine, & Werner, Benjamin. (1993). Synthesis of ML programs in the system Coq. *Journal of symbolic computations*, **15**, 607–640.

Paulson, Lawrence C. (1994). *Isabelle: a generic theorem prover.* Lecture Notes in Computer Science, vol. 828. Springer.

Rosendahl, Mads. (1986). *Automatic program analysis.* M.Phil. thesis, University of Copenhagen.

Rosendahl, Mads. (1989). Automatic complexity analysis. *4th international coneference on functional programming languages and computer architecture.*

Sands, David. (1990). Complexity analysis for a lazy higher-order language. *3rd European symposium on programming.* Lecture Notes in Computer Science, no. 432. Springer.

Sands, David. (1995). A naive time analysis and its theory of cost equivalence. *Journal of logic and computation*, **5**(4).

Smith, Douglas R. (1990). KIDS: A semi-automatic program development system. *IEEE transactions on software engineering*, September.

Spivey, J. Michael. (1992). *The Z notation: A reference manual.* Second edn. International Series in Computer Science. Prentice-Hall.

Wadler, Philip. (1989). Strictness analysis aids time analysis. *Fourth international conference on functional programming languages and computer architecture.*

Wegbreit, Ben. (1975). Mechanical program analysis. *Communications of the ACM*, **18**(9).

Wolfram, Stephen. (1996). *The Mathematica book.* 3 edn.