Combining Automated Theorem Provers and Computer Algebra Systems for Generating Formal Proofs of Complexity Bounds
Extended Abstract

Ralph Benzinger
ralph@cs.cornell.edu
Deptartment of Computer Science
Cornell University
Ithaca, NY 14853, USA

Abstract

Over the past few years, the traditional separation between automated theorem provers and computer algebra systems has slowly been eroded as both sides venture into foreign territory. But despite recent progress, theorem provers still have difficulties with basic arithmetic while computer algebra system inherently produce "untrusted" results that are not easily verified.

We were able to combine successfully two such systems – NUPRL and MATHEMATICA – to build the *Automated Complexity Analysis* (ACA) system for analyzing the computational complexity of higher-order functional programs. The ACA system automatically computes and proves correct an upper bound on the worst-case time complexity of a functional program synthesized by the NUPRL system.

In this extended abstract, we briefly introduce our framework for reasoning informally about the computational complexity of higher-order functional programs and outline our approach to automation. We conclude with a description of employing MATHEMATICA within the trusted NUPRL environment to construct a formal complexity proof.

Keywords: computational complexity, automated complexity analysis, higher-order functional programs, theorem proving, computer algebra systems

Introduction

In a memorable episode during the late 1990s, later to become known as the *Pigeon Hole Incident*, the NUPRL research group at Cornell University was collectively trying to locate the source of inefficiency in a provably correct state-minimization algorithm they had synthesized with the NUPRL proof development system. After an extensive search through the formal proof library that took several weeks, the problem was eventually traced down to an exponential proof of the pigeon hole principle that was cited as a lemma in the main theorem. Once the cause had been found, the running time of the minimization algorithm was quickly made polynomial by rewriting the proof of the lemma.

The Pigeon Hole episode convincingly illustrates the missing link in machine-assisted program synthesis from formal specifications and was one of the main motivations that led to the development of the *Automated Complexity Analysis* (ACA) framework for reasoning about the computational complexity of higher-order functional programs (Benzinger 2001b). The framework applies to any formal system based on operational semantics that can express evaluation within its term language.

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

The ACA System

The current ACA system (Benzinger 2001a; 2001b; 2002) is a reference implementation that automatically computes and proves correct an upper bound on the worst-case time complexity of a functional program synthesized by the NUPRL proof development system (Constable & others 1986). NUPRL is a powerful environment for machine-assisted proof and program development based on computational type theory (Martin-Löf 1983; Constable 1991) and the *proofs-as-programs* principle (Bates & Constable 1985), which facilitates the extraction and execution of computational content from formal proofs.

Examples successfully analyzed with the ACA system include the maximum segment sum problem, several algorithms extracted from different proofs of the pigeon hole theorem, and sorting algorithms that illustrate the differences between call-by-name and call-by-value evaluation strategies.

Notation

In this paper, all terms, including term variables, are set in `typewriter` font, whereas mathematical variables denoting terms or values are set in *italic* font. We will use integer terms and integers referenced by terms interchangeably if the intention will be clear from the context. To save horizontal space in long reduction sequences, we may write $t \downarrow (\text{in } n)\ w$ instead of $t \downarrow w\ (\text{in } n)$.

The Computational Complexity of Functional Programs

While most formal definitions of computational complexity are based on the Turing machine (TM) or the random access machine (RAM) model, modern programming languages featuring advanced data structures or garbage collection increasingly demand for more advanced cost models. This is particularly true for functional languages, whose hidden evaluation cost can be considerable (Lawall & Mairson

1996).

Our framework for defining cost models based on operational semantics is both descriptive and versatile. By selecting different semantics, we can refine our cost measures to get arbitrarily close to a low-level machine model such as the RAM.

Annotated Semantics

To define a complexity measure cc for an operational semantics $\mathcal{S}$ that maps terms to complexity expressions, we annotate each rule $t \downarrow u$ in $\mathcal{S}$ with some complexity information n,

$$t \downarrow^{\mathcal{S}_{\mathrm{cc}}} u \ (\text{in } n),$$

to obtain an *annotated semantics* $\mathcal{S}_{\mathrm{cc}}$. If we can deduce $t \downarrow^{\mathcal{S}_{\mathrm{cc}}} w$ (in n) in $\mathcal{S}_{\mathrm{cc}}$ and w is canonical, we write $t \downarrow\downarrow w$ (in n) and say that t has *value* w and *complexity* n, also denoted by t^* and $\hat{t}$, respectively.

As an example, consider NUPRL's multiplication rule

$$\text{(mul)} \quad \frac{u \downarrow k_1, \quad v \downarrow k_2}{u^\star v \downarrow k_1 k_2}.$$

We can measure *time complexity* by defining

$$\text{(time)} \quad \frac{u \downarrow k_1 \ (\text{in } n_1), \quad v \downarrow k_2 \ (\text{in } n_2)}{u^\star v \downarrow k_1 k_2 \ (\text{in } n_1 + n_2 + 1)},$$

or *space complexity* by defining

$$\text{(space)} \quad \frac{u \downarrow k_1 \ (\text{in } n_1), \quad v \downarrow k_2 \ (\text{in } n_2)}{u^\star v \downarrow k_1 k_2 \ (\text{in } n_1 n_2)},$$

or *"space-time" complexity* by

$$\text{(space-time)} \quad \frac{u \downarrow k_1 \ (\text{in } n_1), \quad v \downarrow k_2 \ (\text{in } n_2)}{u^\star v \downarrow k_1 k_2 \ (\text{in } n_1 + n_2 + \log(k_1 k_2))}.$$

This discussion focuses on the annotated semantics $\mathcal{N}_{\mathrm{time}}$ that measures the time complexity of NUPRL programs with respect to call-by-name evaluation:

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

$$\text{(base)} \quad \frac{m \downarrow \mathtt{0} \ (\text{in } n_1), \quad b \downarrow w \ (\text{in } n_2)}{\mathtt{ind}(m; b; i{,}z.s) \downarrow w \ (\text{in } n_1 + n_2 + 1)}$$

$$\text{(step)} \quad \frac{\begin{array}{c} m \downarrow k > 0 \ (\text{in } n_1), \\ s[k/i, \mathtt{ind}(k{-}1; b; i{,}z.s)/z] \downarrow w \ (\text{in } n_2) \end{array}}{\mathtt{ind}(m; b; i{,}z.s) \downarrow w \ (\text{in } n_1 + n_2 + 1)}$$

For simplicity, $\mathcal{N}_{\mathrm{time}}$ assigns unit cost to each reduction step, but more faithful models using term discrimination or parameterized annotations to incorporate the cost of the evaluator implementation are likewise possible. The interested reader can find the complete definition of $\mathcal{N}_{\mathrm{time}}$ in (Benzinger 2001b).

To reason about the complexity of a given function f, we supply f with *formal arguments* $a_1, \ldots, a_n$ and use the annotated semantics to determine the cost of reducing $f \ \ a_1 \ \ \ldots \ \ a_n$ to canonical form. We define the resulting complexity of this construct to be the complexity of the function f, or more accurately, the complexity of f computing $f(a_1, \ldots, a_n)$ for arbitrary a_i.

As an example, let f be the function term

$$\lambda \mathtt{n.ind(n;} \lambda \mathtt{x,y.x;i,z.}\lambda \mathtt{x,y.z\,y\,(x+y))\ 0\ 1}$$

that computes the n-th Fibonacci number F_n. A simple proof by induction shows that

$$f \ m \downarrow^{\mathcal{N}_{\mathrm{time}}} F_m \ (\text{in } 3m + 3 + F_{m+1} + \hat{m}).$$

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

$$\begin{array}{l} \mathtt{ind}(m; \lambda \mathtt{x,y.x;i,z.}\lambda \mathtt{x,y.z\,y\,(x+y))}\ a\ b \\ \qquad \downarrow F_{m-1}\, a^* + F_m\, b^* \ (\text{in } h(m, \hat{m}, \hat{a}, \hat{b})), \end{array}$$

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

$$\begin{array}{ll} & \mathtt{ind}(m; \lambda \mathtt{x,y.x;i,z.}\lambda \mathtt{x,y.z\,y\,(x+y))}\ a\ b \\ \downarrow & (\text{in } \hat{m}) \\ & \mathtt{ind(0;} \lambda \mathtt{x,y.x;i,z.}\lambda \mathtt{x,y.z\,y\,(x+y))}\ a\ b \\ \downarrow & (\text{in } 1) \\ & (\lambda \mathtt{x,y.x})\ a\ b \\ \downarrow\downarrow & (\text{in } 2 + \hat{a}) \\ & a^*, \end{array}$$

where $F_{-1} = 1$ as usual. For the step case, assume that

$$\begin{array}{ll} & \mathtt{ind}(m-1; \lambda \mathtt{x,y....;i,z.}\lambda \mathtt{x,y....)}\ a\ b \\ \downarrow\downarrow & (\text{in } h(m-1, \widehat{m-1}, \hat{a}, \hat{b})) \\ & F_{m-2}\, a^* + F_{m-1}\, b^* \end{array}$$

Then

$$\begin{array}{ll} & \mathtt{ind}(m; \lambda \mathtt{x,y.x;i,z.}\lambda \mathtt{x,y.z\ y\ (x+y))}\ a\ b \\ \downarrow & (\text{in } 1 + \hat{m}) \\ & (\lambda \mathtt{x,y.ind(}(m-1)^*; \ldots)\ \mathtt{y\ (x+y))}\ a\ b \\ \downarrow & (\text{in } 2) \\ & \mathtt{ind(}(m-1)^*; \ldots)\ \ b\ \ \mathtt{(}a\mathtt{+}b\mathtt{)} \\ \downarrow\downarrow & (\text{in } h(m-1, 0, \hat{b}, \widehat{a+b})) \\ & F_{m-2}\, b^* + F_{m-1}(a+b)^*. \end{array}$$

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

$$\begin{array}{ll} & 1 + \hat{m} + 2 + h(m-1, 0, \hat{b}, \widehat{a+b}) \\ = & 3 + \hat{m} + 3(m{-}1) + 2 + F_m + F_{m-2}\,\hat{b} + F_{m-1}\,\widehat{a+b} \\ = & 2 + \hat{m} + 3m + F_m + F_{m-2}\,\hat{b} + F_{m-1}(\hat{a} + \hat{b} + 1) \\ = & 2 + \hat{m} + 3m + (F_m + F_{m-1}) + (F_{m-1} + F_{m-2})\,\hat{b} \\ & \quad + F_{m-1}\,\hat{a} \\ = & h(m) \end{array}$$

The original claim now follows trivially from $\hat{\mathtt{0}} = \hat{\mathtt{1}} = 0$. □

Higher-Order Complexity

The major challenge in developing a formal calculus for higher-order complexity is to find the right means to refer to the complexity of unknown terms while maintaining compositionality. To describe the complexity of a second-order term such as

$$(\lambda \mathtt{g.g\ 0})\ f,$$

where f is a formal argument of type $\mathbb{Z} \to \mathbb{Z}$, an expression like $2 + \hat{f} + \widehat{f\texttt{*0}}$ is intuitively meaningful but of limited value. Obviously, unqualified use of the hat notation does not provide any insight into the computational behavior of a program, as $X \downarrow\downarrow X^*$ (in $\hat{X}$) is vacuously true for even the most complicated computation X.

To overcome this limitation, we use *type decomposition* to characterize the computational behavior of f without quoting the actual term structure. More precisely, we break higher-order arguments into the values and complexities of their constituent subtypes, which, once identified, can be uniformly referred to within the calculus. This 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 (Tait 1967).

Type decomposition as defined in (Benzinger 2001b) is compositional with respect to value and complexity. The structure of the decomposition depends uniformly on type. Given a formal argument a of some type τ, we associate with τ a list of higher-order functions f_i that express the values and complexities of evaluating instantiations of a while abstracting from the actual term structure of a or any of its formal inputs a_i. This *abstract type decomposition* can then be used to reason meaningfully about the computational behavior of a.

In general, the decomposition of a term or formal argument a of type $\tau_1 \to \cdots \to \tau_n \to \mathbb{Z}$ is a list of $2n + 2$ concrete or abstract functions

$$\begin{array}{rcl} & \mapsto & f_{0v}, f_{0c} \\ d_1 & \mapsto & f_{1v}(d_1), f_{1c}(d_1) \\ & \cdots & \\ d_1, \ldots, d_n & \mapsto & f_{nv}(d_1, \ldots, d_n), f_{nc}(d_1, \ldots, d_n), \end{array}$$

where each d_i is the decomposition of formal argument a_i: τ_i for a. As a simple example, the decomposition of the first-order function $\lambda\texttt{x.x*x}$ of type $\mathbb{Z} \to \mathbb{Z}$ is

$$(\lambda\texttt{x.x*x},\ 0,\ (x_v, x_c) \mapsto x_v^2,\ (x_v, x_c) \mapsto 2x_c + 1),$$

representing the fact that (x_v, x_c) is the abstract decomposition of formal argument x: $\mathbb{Z}$ and $\lambda\texttt{x.x*x} \downarrow\downarrow \lambda\texttt{x.x*x}$ (in 0) and $(\lambda\texttt{x.x*x})\ x \downarrow\downarrow (x^*)^2$ (in $2\hat{x} + 1$).

As the type decomposition of higher-order terms comprises higher-order functions, their complexity is naturally expressed using higher-order polynomials (Irwin, Kapron, & Royer 2000). This solution, however, is at odds with the prevailing first-order notion of complexity that is deeply rooted in imperative languages and the random access machine model. We thus use *polynomialization* to express higher-order complexity in first-order terms wherever possible.

Symbolic Evaluation

Formal arguments are *meta variables* of our term language representing arbitrary type-correct terms. Consequently, any expression involving formal arguments is not a proper term and thus not understood by the evaluation system.

To automate our informal complexity analysis described earlier in this paper, we need to extend $\mathcal{N}_{\text{time}}$ to a *symbolic semantics* $\bar{\mathcal{N}}_{\text{time}}$ in which the meta variables of $\mathcal{N}_{\text{time}}$ become proper terms. This extension involves *symbolic terms* that soundly represent classes of proper terms in $\mathcal{N}_{\text{time}}$.

The central building block of $\bar{\mathcal{N}}_{\text{time}}$ or any other symbolic semantics is the symbolic `cpx*` term used to represent a class of terms that exhibit a certain computational behavior:

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

The meta variables v of type τ of $\mathcal{N}_{\text{time}}$ are encoded as parameterized symbolic terms $\texttt{var*(}v; \tau\texttt{)}$. The reduction rules for `var*` automate our idea of the abstract type decomposition by introducing a type-correct construct of `cpx*` terms that provides canonical names for the decomposition functions f_{iv} and f_{ic}.

Solving Higher-Type Recurrences with MATHEMATICA

The automatic analysis of primitive or general recursive functions generates systems of parameterized higher-type recurrence equations (REs)

$$R(p_1, \ldots, p_n, m) = \begin{cases} F(p_1, \ldots, p_n) & \text{if } m = 0 \\ G(p_1, \ldots, p_n, m, R) & \text{if } m > 0 \end{cases}$$

for which we require *closed solutions*. Although no complete algorithm for finding closed solutions to arbitrary REs exists, we can identify many important subclasses such as linear equations with constant coefficients or hypergeometric equations for which complete algorithms have been found. The *finite calculus* and the *method of generating functions* in particular have proven successful for linear REs.

All major computer algebra systems, including MATHEMATICA, MAPLE, and MUPAD, provide some functionality for solving general univariate RE systems, usually by means of generating functions. Unfortunately, none of these systems supports multivariate REs. Instead of reinventing the wheel by creating a custom recurrence solver, we reduce the parameterized higher-type REs generated during complexity analysis to unparameterized simple-type REs. Our resulting algorithm for parameterized REs is generally applicable and not limited to equations generated by the ACA system. The procedure is necessarily incomplete and cannot obtain closed solutions for all possible REs, including those generated by ACA.

Formal Proof Construction

The complexity results returned by the symbolic evaluator are *untrusted* in the sense that their correctness depends critically on the correctness of the evaluator as a whole. This is an undesirable yet very common situation shared by most commercial software tools, which rely on traditional means of quality control—most notably testing—to build confidence in their results.

Theorem provers like NUPRL, on the other hand, produce *trusted* results that are provably correct short of actual defects in the computing environment. In most of these systems, the proof is constructed by a large, untrusted proof

generator and then verified by a small, trusted proof checker whose correctness has been established once and for all by external means. In the case of NUPRL, this trusted core is a simple type checking algorithm of several hundred lines of LISP code.

The proof generator of the ACA system constructs a formal NUPRL proof that asserts the correctness of the informal complexity result obtained by the symbolic evaluator. As all such complexity theorems are fully integrated into NUPRL's standard library, the NUPRL proof checker automatically ensures their validity.

The use of symbolic algebra systems within a theorem prover is a novel approach to guarantee the correctness of these untrusted systems. A similar yet opposite approach is taken by THEOREMA (Buchberger *et al.* 1997) and ANALYTICA (Clarke & Zhao 1992), which implement a theorem prover within a symbolic algebra system.

Formal Evaluation

The equality type built into the NUPRL system is *extensional* in that terms such as `6*9` and `42` that reduce to the same value are interchangeable in any context. Alas, this property makes it impossible to reason about terms as opposed to their values in the standard theory.

To address this situation within the ACA system, we adopted one of several proposals for *term reflection* (Constable & Crary 2001). Let $\mathbb{T}_1$ denote the reflected terms of NUPRL. For the purpose of this presentation, the reader might think of a reflected term as a term with a special tag ($^{\#}$) that prevents the identification of extensionally but not intensionally equal terms. For each type τ, the reflected type $[\tau]$ comprises the reflected terms of all terms in τ. If t is a reflected term in $[\tau]$, we write $[\![t]\!]$ for the corresponding unreflected term in τ.

We define a function $\mathtt{red}\colon \mathbb{T}_1 \to \mathbb{T}_1 + \mathtt{Unit}$ that performs one reduction step if possible and returns $\mathtt{inr}(\diamond)$ otherwise. We write $t_1 \mapsto t_2$ if and only if $\mathtt{red}(t_1)$ returns $\mathtt{inl}(t_2)$. We can then formalize the evaluation predicate $t_1 \downarrow^{\mathcal{N}_{\text{time}}} t_2$ (in n) within NUPRL's logic:

$$\begin{aligned} t_1 \downarrow t_2 \text{ (in } n) \;==\; & t_1 = t_2 \in \mathbb{T}_1 \;\vee \\ & n \neq 0 \;\&\; t_1 \mapsto t_2 \;\vee \\ & \exists u\colon \mathbb{T}_1.\ \exists n_1, n_2\colon \mathbb{N}.\ n_1 + n_2 \leq n \;\& \\ & \qquad t_1 \downarrow u \text{ (in } n_1) \;\&\; u \downarrow t_2 \text{ (in } n_2). \end{aligned}$$

For convenience, we define two additional predicates

$$\begin{aligned} & t_1 \downarrow\downarrow t_2 \text{ (in } n) \;== \\ & \quad t_1 \downarrow t_2 \text{ (in } n) \;\&\; \mathtt{red}(t_2) = \mathtt{inr}(\diamond) \in \mathbb{T}_1 + \mathtt{Unit} \\ & t_1 \downarrow\downarrow \text{ (in } n) \;== \\ & \quad \exists t_2\colon \mathbb{T}_1.\ t_1 \downarrow\downarrow t_2 \text{ (in } n) \end{aligned}$$

that make the statement of complexity results more succinct.

While we are free to decompose above predicates in any way we like, the proof tree of a general complexity judgment

$$t_1\colon \mathbb{T}_1,\ t_2\colon \mathbb{T}_1,\ n\colon \mathbb{N} \;\vdash\; t_1 \downarrow t_2 \text{ (in } n)$$

will usually be an exact mirror image of the symbolic evaluation tree for $t_1 \downarrow t_2$ (in n). The ACA proof generator translates the tree for $t[\bar{a}] \downarrow w$ (in n) with formal arguments $a_i\colon \tau_i$ into a formal NUPRL theorem and a corresponding proof tactic that proves it.

A Formal Example

The second-order Fibonacci function

```
fibo ==
    λn.ind(n;λf.f 0 1;
            i,z.λf.z (λx,y.f y (x+y)))
        λx,y.x
```

has complexity $\mathtt{fibo}\ m \downarrow\downarrow$ (in $4m + 5 + \hat{m} + F_{m+1}$) as inferred by the symbolic evaluator. Stated formally, this claim becomes

$$\begin{aligned} \vdash\ & \forall \mathtt{m}, \mathtt{m}_v\colon [\mathbb{N}].\ \forall \mathtt{m}_c\colon \mathbb{N}.\ \mathtt{m} \downarrow\downarrow \mathtt{m}_v \text{ (in } \mathtt{m}_c) \supset \\ & \quad \mathtt{apply}^{\#}(\mathtt{fibo}^{\#}; \mathtt{m}) \downarrow\downarrow \\ & \qquad \text{(in } 4\,[\![\mathtt{m}]\!] + 5 + F_{[\![\mathtt{m}]\!]+1} + \mathtt{m}_c), \end{aligned}$$

where F_n is any integer term denoting the n-th Fibonacci number. The main proof generated by the system references a lemma that characterizes the behavior of the inductive term

$$\mathtt{ind}(\mathtt{m}; \lambda\mathtt{f}.\mathtt{f}\ 0\ 1; \mathtt{i},\mathtt{z}.\lambda\mathtt{f}.\mathtt{z}\ (\lambda\mathtt{x},\mathtt{y}.\mathtt{f}\ \mathtt{y}\ (\mathtt{x}+\mathtt{y})))$$

used in the definition of `fibo`:

$$\begin{aligned} \vdash\ & \forall \mathtt{m}\colon [\mathbb{N}].\ \exists \mathtt{r}_v\colon [(\mathbb{N} \to \mathbb{N} \to \mathbb{N}) \to \mathbb{N}]. \\ & \quad \mathtt{ind}^{\#}(\mathtt{m}; \lambda^{\#}\mathtt{f}^{\#}\ldots; \mathtt{i}^{\#},\mathtt{z}^{\#}.\lambda^{\#}\mathtt{f}^{\#}\ldots) \downarrow\downarrow \mathtt{r}_v \text{ (in } 1) \\ & \quad \&\ \forall \mathtt{r}_i, \mathtt{r}_{iv}\colon [\mathbb{N} \to \mathbb{N} \to \mathbb{N}]. \\ & \qquad \forall \mathtt{r}_{ic}, \mathtt{r}_{io1}, \mathtt{r}_{io0}, \mathtt{r}_{ioo10}, \mathtt{r}_{ioo01}, \mathtt{r}_{ioo00}\colon \mathbb{N}. \\ & \qquad\quad \delta \supset \exists \mathtt{r}_{ov}\colon [\mathbb{N}].\mathtt{apply}^{\#}(\mathtt{r}_v; \mathtt{r}_i) \downarrow\downarrow \mathtt{r}_{ov} \\ & \qquad\qquad \text{(in } 4\,[\![\mathtt{m}]\!] + 3 + \mathtt{r}_{ic} + \mathtt{r}_{io0} + \mathtt{r}_{ioo00} \\ & \qquad\qquad\quad + (F_{\mathtt{m}+1} - 1)(\mathtt{r}_{io1} + \mathtt{r}_{ioo10}) \\ & \qquad\qquad\quad + (F_{\mathtt{m}+2} - 1)\mathtt{r}_{ioo01}), \end{aligned}$$

where δ describes the computational behavior of formal input argument r_i of type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$:

$$\begin{aligned} \delta\ ==\ & \mathtt{r}_i \downarrow\downarrow \mathtt{r}_{iv} \text{ (in } \mathtt{r}_{ic}) \\ & \&\ \forall \mathtt{r}_{ii}, \mathtt{r}_{iiv}\colon [\mathbb{N}].\ \forall \mathtt{r}_{iic}\colon \mathbb{N}. \\ & \quad \mathtt{r}_{ii} \downarrow\downarrow \mathtt{r}_{iiv} \text{ (in } \mathtt{r}_{iic}) \\ & \quad \supset \exists \mathtt{r}_{iov}\colon [\mathbb{N} \to \mathbb{N}]. \\ & \qquad \mathtt{apply}^{\#}(\mathtt{r}_{iv}; \mathtt{r}_{ii}) \downarrow\downarrow \mathtt{r}_{iov} \text{ (in } \mathtt{r}_{io0} + \mathtt{r}_{io1}\,\mathtt{r}_{iic} + 1) \\ & \qquad \&\ \forall \mathtt{r}_{ioi}, \mathtt{r}_{ioiv}\colon [\mathbb{N}].\ \forall \mathtt{r}_{ioic}\colon \mathbb{N}. \\ & \qquad\quad \mathtt{r}_{ioi} \downarrow\downarrow \mathtt{r}_{ioiv} \text{ (in } \mathtt{r}_{ioic}) \\ & \qquad\quad \supset \exists \mathtt{r}_{ioov}\colon [\mathbb{N}]. \\ & \qquad\qquad \mathtt{apply}^{\#}(\mathtt{r}_{iov}; \mathtt{r}_{ioi}) \downarrow\downarrow \mathtt{r}_{ioov} \\ & \qquad\qquad\quad \text{(in } \mathtt{r}_{ioo00} + \mathtt{r}_{ioo01}\,\mathtt{r}_{ioic} + \mathtt{r}_{ioo10}\,\mathtt{r}_{iic} + 1) \end{aligned}$$

The recursive character of the lemma suggests a proof by natural induction on `m`. The main difficulty in developing such a proof is to derive appropriate inductive hypotheses for the (in ...) bounds of the statement. Fortunately, these bounds are determined by the symbolic evaluator, so the automated construction of the proof becomes straightforward. The complete proof tree for above example can be found in (Benzinger 2001b) or the ACA system distribution.

Conclusion

We have shown how we can combine automated theorem provers and computer algebra systems for creating formal proofs of complexity bounds. The resulting system is provably correct in the sense of the theorem proving community yet harnesses the full power of a computer algebra system.

We believe that by improving the interface between NUPRL and MATHEMATICA, we will be able not only to broaden the scope of NUPRL's arithmetical tactics but also to formalize some parts of MATHEMATICA's computation process.

References

Bates, J. L., and Constable, R. L. 1985. Proofs as programs. *ACM Transactions on Programming Languages and Systems* 7(1):113–136.

Benzinger, R. 2001a. Automated complexity analysis of NUPRL extracted programs. *Journal of Functional Programming* 11(1):3–31.

Benzinger, R. 2001b. *Automated Computational Complexity Analysis*. Ph.D. Dissertation, Cornell University.

Benzinger, R. 2002. Automated higher-order complexity analysis. *Theoretical Computer Science*. Accepted for publication.

Buchberger, B.; Jebelean, T.; Kriftner, F.; Marin, M.; Tomuţa, E.; and Vāsaru, D. 1997. A survey of the theorema project. In Küchlin, W. W., ed., *ISSAC '97. Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, July 21–23, 1997, Maui, Hawaii*, 384–391. New York, NY 10036, USA: ACM Press.

Clarke, E., and Zhao, X. 1992. Analytica — A theorem prover in Mathematica. *Lecture Notes in Computer Science* 607:761–765.

Constable, R. L., and Crary, K. 2001. Computational complexity and induction for partial computable in type theory. In Sieg, W.; Sommer, R.; and Talcott, C., eds., *Reflections: A collection of essays in honor of Solomon Feferman*, Lecture Notes in Logic. Association for Symbolic Logic. Expected publication June 2001.

Constable, R. L., et al. 1986. *Implementing Mathematics with the* NUPRL *Proof Development System*. Prentice-Hall.

Constable, R. L. 1991. Type theory as a foundation for computer science. *Lecture Notes in Computer Science* 526.

Irwin, R. J.; Kapron, B. M.; and Royer, J. S. 2000. Separating notions of higher-type polynomial time. In *Proceedings of the Implicit Computational Complexity Workshop*.

Lawall, J. L., and Mairson, H. G. 1996. Optimality and efficiency: What isn't a cost model of the lambda calculus? In *ACM International Conference on Functional Programming*, 92–101.

Martin-Löf, P. 1983. On the meanings of the logical constants and the justifications of the logical laws. Unpublished manuscript. Department of Mathematics, University of Stockholm.

Tait, W. W. 1967. Intensional interpretation of functionals of finite type. *J. Symbolic Logic* 32:198–212.