SUBSTRUCTURAL VERIFICATION AND COMPUTATIONAL FEASIBILITY*

Daniel Leivant
Indiana University

Abstract We refer to the intrinsic theories of [14, 17], a generic framework for uncoded reasoning about equational programs. In particular, a natural notion of provable functions corresponds to the provably recursive functions of Peano Arithmetic and similar systems. A natural-deduction formulation of these systems map directly, via a Curry-Howard morphism, to terms of the simply typed lambda calculus with recurrence, with a termination proof for a function f mapping to a representation of f.

In [16] we showed that natural structural restrictions on derivations correspond to major complexity classes. When induction is restricted to positive formulas, a generalization of Σ_1^0 formulas, exactly the primitive recursive functions are provable. When only a "predicative" form of induction is allowed we obtain the Kalmar elementary functions. The combination of both restrictions yields the functions computable in polynomial time.

We show here that induction over arbitrary formulas does not add new provable functions if we disallow in derivations the closing of multiple data-complex assumptions. This significantly extends the class of proofs that can be accepted as "feasible mathematics."

We also show that if multiple closing of data-complex assumptions is only prohibited when above distinct premises of implication elimination, then the provable functions are precisely the functions computable in polynomial space.

Keywords: Implicit computational complexity, proof theory, substructural proofs, intrinsic theories, program verification, ramified induction, equational programs, program termination, feasibility, polynomial time, polynomial space, elementary functions, typed lambda calculi.

*Research partially supported by NSF grant CCR-0105651.

1. BACKGROUND

1.1. Intrinsic theories

In [14, 17] we introduced a verification methodology for equational programs, dubbed *intrinsic theories*. For each inductively-generated data system C one uses a skeletal theory $\mathbf{IT}(C)$, whose axioms are merely data-introduction axioms, i.e. the closure of data under the basic constructors, and data-elimination, i.e. induction schemas for the data types. To keep this condensed presentation uncluttered, we focus on the term algebra most relevant to computational complexity, namely the algebra $\mathbb{W}$ generated from the constant ε and the unary constructors $\mathbf{0}$ and $\mathbf{1}$, i.e. essentially the set $\{0,1\}^*$. The axioms of $\mathbf{IT}(\mathbb{W})$ are then the data-introduction axioms, which we supplement with a destructor rule, and write as inference rules:

$$\frac{}{\mathsf{W}(\varepsilon)} \qquad \frac{\mathsf{W(t)}}{\mathsf{W(it)}} \qquad \frac{\mathsf{W(it)}}{\mathsf{W(t)}} \qquad (i=0,1)$$

and data-elimination, i.e. the induction schema

$$\frac{\mathsf{W}(t) \quad \varphi[\varepsilon] \quad \varphi[z]\rightarrow\varphi[\mathbf{0}z] \quad \varphi[z]\rightarrow\varphi[\mathbf{1}z]}{\varphi[t]}$$

with z not free in open assumptions.[1] A *degenerated form* of data-elimination is

$$\frac{\mathsf{W}(t) \quad \varphi[\varepsilon] \quad \varphi[\mathbf{0}x] \quad \varphi[\mathbf{1}x]}{\varphi[t]}$$

i.e. reasoning by cases.

Throughout the paper we refer to provability in *intuitionistic logic*.[2] Moreover, we greatly simplify the discussion by referring to the fragment of logic without disjunction and $\exists$.

1.2. Provable equational programs

We refer to equational programs over $\mathbf{IT}(\mathbb{W})$. Each such program consists of a finite set P of equations between terms, where the terms are built from variables, the constructors ε, $\mathbf{0}$ and $\mathbf{1}$, and program function-identifiers. One identifier is singled out as the program's principal identifier. If $\mathbf{f}$ is the principal identifier of P, we say that $(P,\mathbf{f})$ *computes* a function f over $\mathbb{N}$ if $f(\vec{n}) = m$ exactly when the formal equation $\mathbf{f}(\vec{n}) = m$ is derived from P in equational

[1]See [17] for the generic rules. It is also natural to consider *separation axioms*, which guarantee that the denotation of all ground terms are distinct; for $\mathbb{N}$ these are Peano's third and fourth axioms, $\forall x.\ \mathrm{s}x \neq 0$ and $\forall x,y.\ \mathrm{s}x = \mathrm{s}y \rightarrow x = y$. However, these axioms have no effect on the provability of programs, as defined below; see [16].

[2]Indeed, the calibration of proofs' computational contents by structural conditions is more problematic and less rewarding when classical logic is used; compare [16, §3.3].

logic. A program P with principal r-ary function identifier $\mathbf{f}$, is *provable* (over a given logic $\mathbf{L}$) if

$$\mathbf{IT}(\mathbb{W}), \forall P, \mathsf{W}(x_1)\ldots\mathsf{W}(x_r) \vdash \mathsf{W}(\mathbf{f}(\vec{x}))$$

where $\forall P$ is the universal closure of the conjunction of P, and provability is in $\mathbf{L}$.

Two examples of provable programs, which will be of use later, are addition and multiplication over $\mathbb{W}$, defined by $+\varepsilon x = x$, $+\mathsf{i}yx = \mathbf{i}(+yx)$, $*x\varepsilon = \varepsilon$, and $*x(\mathsf{i}y) = +x(*xy)$ $(i = 0, 1)$. Here are derivations for these two functions, where we use double-bars for the contraction of trivial steps, and display generically the induction cases for the successor functions $\mathbf{0}$ and $\mathbf{1}$.

$$\dfrac{\mathsf{W}(y) \qquad \dfrac{\dfrac{\forall x.\ +\varepsilon x = x}{x = +\varepsilon x} \quad \mathsf{W}(x)}{\mathsf{W}(+\varepsilon x)} \qquad \dfrac{\dfrac{\dfrac{\forall P}{\mathsf{i}(+yz) = +(\mathsf{i}y)z} \quad \dfrac{\overset{(1)}{\mathsf{W}(+yz)}}{\mathsf{W}(\mathsf{i}(+yz))}}{\mathsf{W}(+(\mathsf{i}y)z)}}{\mathsf{W}(+yz) \to \mathsf{W}(+(\mathsf{i}y)z)}\,(1)}{\mathsf{W}(+xy)}$$

$$\dfrac{\mathsf{W}(y) \qquad \dfrac{\dfrac{\forall P}{\varepsilon = *x\varepsilon} \quad \mathsf{W}(\varepsilon)}{\mathsf{W}(*x\varepsilon)} \qquad \dfrac{\dfrac{\dfrac{\forall P}{+x(*xz) = *x(\mathsf{i}z)} \quad \dfrac{\mathsf{W}(x) \quad \overset{(1)}{\mathsf{W}(*xz)}}{\mathsf{W}(+x(*xz))}\mathcal{D}}{\mathsf{W}(*x(\mathsf{i}z))}}{\mathsf{W}(*xz) \to \mathsf{W}(*x(\mathsf{i}z))}\,(1)}{\mathsf{W}(*xy)}$$

where $\mathcal{D}$ is the derivation above for addition, with $*xz$ substituted for the free occurrences of y.

1.3. Morphism to λ-terms

Let λ_1 be the (Church-style) simply-typed lambda calculus defined as follows. The *types* are generated from base types ι (for elements of $\mathbb{W}$) and θ (a unit type), using the binary type operations $\to$ and $\times$. We call arrow-free types *positive*. For all type τ we identify all of $\theta \times \tau$, $\tau \times \theta$, and $\theta \to \tau$ with τ; also, we identify $\tau \to \theta$ with θ. In each case we say that the shorter form is a contraction of the longer one, and we say that τ' is the *contracted form of* τ if τ' is obtained from τ by successive contractions, and cannot be contracted further (i.e. is either θ or free of θ).

We omit parentheses when in no danger of ambiguity, modulo the proviso that $\times$ binds stronger than $\to$, and then that $\to$ and $\times$ associate to the right. For example, $\iota \to \iota \times \iota \to \iota$ abbreviates $\iota \to ((\iota \times \iota) \to \iota)$. We call a type *positive* if its contracted form is free of $\to$.

For each type τ we posit an unbounded stock of *variables* of type τ, x_i^τ (we omit the type superscript when convenient). Terms are generated from the variables using λ-abstraction, type-correct application, pairing (written $\langle E_0, E_1\rangle$), and type-correct projection (written $\pi_i E$, $i = 0$ or 1). The corresponding types are defined as usual. We write $\langle E_0, \ldots, E_m\rangle$ for $\langle E_0, \langle E_1, \cdots, \langle E_{m-1}, E_m\rangle \cdots\rangle\rangle$. The computational rules are β-reduction and projection-reduction. We write $E \Rightarrow E'$ (and say that E converts to E') if E' arises by replacing in E a subterm F by its reductum.

The *typed lambda calculus over* $\mathbb{W}$, $\lambda_1(\mathbb{W})$, is the extension of λ_1 with constants $*$ of type θ, ε of type ι, $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{p}$ of type $\iota \to \iota$, $\mathbf{B}$ (branching) of type $(\iota, \iota \to \iota, \iota \to \iota, \iota) \to \iota$, and for each type τ $\mathbf{R}_\tau$ of type $\tau \to (\tau \to \tau) \to \iota \to \tau$. The reduction rules of λ_1 are augmented with:

$$\begin{array}{rcl} \mathbf{p}(\mathbf{i}\mathbf{t}) & \to & \mathbf{t} \qquad (i = 0, 1) \\[6pt] \mathbf{B}\mathbf{t}_\epsilon \mathbf{t}_0 \mathbf{t}_1 \varepsilon & \to & \mathbf{t}_\epsilon \\ \mathbf{B}\mathbf{t}_\epsilon \mathbf{t}_0 \mathbf{t}_1(\mathbf{i}\mathbf{w}) & \to & \mathbf{t}_i(\mathbf{w}) \quad (i = 0, 1) \\[6pt] \mathbf{R}\mathbf{t}_\epsilon \mathbf{t}_0 \mathbf{t}_1 \varepsilon & \to & \mathbf{t}_\epsilon \\ \mathbf{R}\mathbf{t}_\epsilon \mathbf{t}_0 \mathbf{t}_1(\mathbf{i}\mathbf{w}) & \to & \mathbf{t}_i(\mathbf{R}\mathbf{t}_\epsilon \mathbf{t}_0 \mathbf{t}_1 \mathbf{w}) \end{array}$$

We define a mapping κ from $\mathbf{IT}(\mathbb{W})$ formulas to types of $\lambda_1(\mathbb{W})$:[3]

$$\begin{array}{rcll} \kappa(E) & = & \theta & \text{if } E \text{ is an equation} \\ \kappa(\mathbf{W}(\mathbf{t})) & = & \iota & \\ \kappa(\varphi_0 \wedge \varphi_1) & = & \kappa(\varphi_0) \times \kappa(\varphi_1) & \\ \kappa(\varphi_0 \to \varphi_1) & = & \kappa(\varphi_0) \to \kappa(\varphi_1) & \\ \kappa(\forall x \varphi) & = & \kappa(\varphi) & \end{array}$$

Thus, κ extracts from a formula φ the type $\kappa\varphi$ of its "computational contents." We extend κ to a Curry-Howard mapping from derivations $\mathcal{D}$ of $\mathbf{IT}(\mathbb{W})$ to to terms $\kappa\mathcal{D}$ of $\lambda_1(\mathbb{W})$. If $\mathcal{D}$ derives a formula φ from labeled assumptions $\overset{(\ell_i)}{\psi_i}$ then $\kappa\mathcal{D}$ will be a term of type $\kappa\varphi$, with free variables $x_{\ell_i}^{\kappa\psi_i}$. The definition of κ is given in a table at the end of the paper.[4] The mapping κ is a homomorphism with respect to reductions:[5] if $\mathcal{D}$ reduces to $\mathcal{D}'$, then $\kappa\mathcal{D}'$ is either identical to $\kappa\mathcal{D}$ or is obtained from it by a reduction in $\lambda_1(\mathbb{W})$.

THEOREM 1 [14, 17, 16] *(1) If $\mathcal{D}$ is a proof of* $\mathbf{IT}(\mathbb{W})$ *for* $(P, \mathbf{f})$ *then* $\kappa\mathcal{D}$ *is a program of* $\lambda_1(\mathbb{W})$ *for the function computed by* $(P, \mathbf{f})$.

[3]The oblivion of this mapping to terms and first order quantifiers was first used in [13]. The unit type was first used in this context in [17].

[4]The definition of κ for data rules (i.e. those referring to $\mathbf{N}$) is shorter when the latter are formulated as axioms, but we prefer to keep them as as inference rules, which offer a more streamlined proof theoretic treatment of normalization, as well as of structural conditions on induction.

[5]See [17] for the definition of reduction for the data-rules.

(2) The provable functions of **IT**(𝕎) *are precisely (modulo canonical codings) the provably-recursive functions of Peano Arithmetic.*

(3) The functions provable in **IT**(𝕎) *with induction for positive formulas are the primitive recursive functions.* ⊣

Part (1) is a useful tool here and in similar formalism: it enables one to focus attention on the computationally relevant aspects of proofs, namely those that are coded in the corresponding $\lambda_1(\mathbb{W})$-terms. For example, the fact that every function provable using positive induction is primitive recursive immediately follows from (1), since such proofs map under κ to terms of $\lambda_1(\mathbb{W})$ with first-order recurrence. Similarly, (2) follows from (1) by [7].

1.4. Predicative induction

Ed Nelson [20] and others have noticed that first order arithmetic has an impredicative ingredient, responsible for the admission of unfeasible functions such as exponentiation. This implicit impredicativity is clearly identified in the induction rule of **IT**(𝕎): from $\mathbf{W}(\mathbf{t})$ one derives $\varphi[\mathbf{t}]$ in which $\mathbf{W}$ itself may occur. Viewing induction as delineating 𝕎 (or — similarly — ℕ), is therefore circular.

Intrinsic theories are useful not only for articulating this impredicativity, but also for addressing it. One such method is ramification [14], which is analogous to the ramification of second order logic to break its impredicativity [21]. Combinatorially speaking, ramification of induction blocks an exponential explosion of proof normalization by preventing that the major premise of induction depend on an induction hypothesis of another induction. A more direct blocking was defined in [16]. Call a labeled assumption φ (in a natural deduction derivation $\mathcal{D}$) a ***working assumption*** if it is closed in $\mathcal{D}$. Call a formula φ ***data significant*** if the contracted form of $\kappa\varphi$ is not θ. Call an instance of induction ***predicative*** if its major (i.e. leftmost) premise does not depend on an open data-siginficant working assumption. A derivation is ***predicative*** if all non-degenerated instances of induction are predicative.

THEOREM 2 [16] *A function over* 𝕎 *is computable in polynomial time iff it is computed by an equational program* $(P, \mathbf{f})$ *which s provable in* **IT**(𝕎) *by a predicative derivation with induction for data-positive formulas.*

The proof uses the homomorphism κ: if $\mathcal{D}$ is a derivation of $(P, \mathbf{f})$ with only predicative instances of induction, and all induction formulas data-positive, then $\kappa\mathcal{D}$ is a $\lambda_1(\mathbb{W})$-term with $\mathbf{R}_\tau$ for positive τ only, where all instances of recurrence are ***predicative***, i.e. with no variable both free in the recurrence argument and bound in $\kappa\mathcal{D}$.

2. SOLITARY INFERENCES AND POLY-TIME

2.1. Multiple uses of assumptions

We illustrate the power of induction for non-positive formulas by proving a function over $\mathbb{N}$ of exponential growth rate, namely the function $e(x,y) = 2^x + y$ defined by the program consisting of the equations $\mathsf{e}(\mathbf{0},y) = \mathsf{s}y$, $\mathsf{e}(\mathsf{s}x,y) = \mathsf{e}(x,\mathsf{e}(x,y))$. Here is a natural deduction proof for this program, in $\mathbf{IT}(\mathbb{N})$, the intrinsic theory for $\mathbb{N}$ analogous to $\mathbf{IT}(\mathbb{W})$.

$$\dfrac{\mathsf{N}(x) \quad \dfrac{\dfrac{\dfrac{\mathsf{N}(y)}{\mathsf{N}(\mathsf{s}y)}}{\mathsf{N}(\mathsf{e}(0,y))}}{\forall y\,(\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(0,y)))} \quad \dfrac{\dfrac{\dfrac{\dfrac{\forall y\,(\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(u,y)))}{\mathsf{N}(\mathsf{e}(u,y))\to\mathsf{N}(\mathsf{e}(u,\mathsf{e}(u,y)))} \quad \dfrac{\dfrac{\forall y\,(\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(u,y)))}{\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(u,y))} \quad \mathsf{N}(y)}{\mathsf{N}(\mathsf{e}(u,y))}}{\mathsf{N}(\mathsf{e}(u,\mathsf{e}(u,y)))}}{\mathsf{N}(\mathsf{e}(\mathsf{s}u,y))}}{\forall y\,(\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(\mathsf{s}u,y)))}}{\forall y\,(\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(x,y)))}\ ind$$

The fact that the induction formula here is not positive is exploited by using it twice, and applying the outcome of one use to the outcome of the other. The question arrises then: is the culprit for exponential growth-rate the very complexity of the induction formula, or merely the duplicate use made of it? The answer is the latter.

2.2. Solitary assumption-classes

Theorem 2 shows that powerful restrictions on proof methods yield poly-time complexity. We show that these restrictions can be relaxed without yielding additional provable functions (though possibly proving additional *programs* for such functions). We start by presenting the main idea in its simplest form.

A labeled-assumption in a derivation $\mathcal{D}$ is *solitary* if it has at most one formula-occurrence.[6] A derivation is *strictly-solitary* if all its assumption classes are solitary.

If $\mathcal{D}$ is a strictly-solitary derivation, then the $\lambda_1(\mathbb{W})$-term $\kappa\mathcal{D}$ has the property that every λ-abstraction closes at most one variable-occurrence. Let M be such a term, and suppose that M reduces to M'. The two salient properties of the reduction are: (1) all λ-abstraction in M' again close at most one variable-occurrence; and (2) the size of M' is smaller than the size M (though the height of M' may be roughly double that of M).

PROPOSITION 3 *A function over* $\mathbb{W}$ *is computable in polynomial time iff it is computed by an equational program* $(P,\mathbf{f})$ *which is provable in* $\mathbf{IT}(\mathbb{W})$ *by a strictly-solitary derivation* $\mathcal{D}$.

[6]Recall that an assumption-class is the set of commonly-labeled assumptions in a natural deduction derivation, closed jointly by some inference.

Notice that in the derivation $\mathcal{D}$ for a program $(P, \mathbf{f})$ the assumptions $\mathsf{W}(x_i)$ are not closed, and therefore may have multiple occurrences. Also note that we make no stipulation about the complexity of induction formulas or their predicative use.

Proof Outline. The proof of Theorem 2 shows that all poly-time functions have a program with a strictly-solitary derivation $\mathcal{D}$.

To prove the converse, consider a strictly-solitary derivation $\mathcal{D}$ for a program $(P, \mathbf{f})$ computing the function f. Then $M \equiv \kappa\mathcal{D}$ is a $\lambda_1(\mathbb{W})$-term that computes f, and in which no λ-abstraction closes more than one variable occurrence. Consider a term Mw, where $w \in \mathbb{W}$. By induction on M and secondary induction on $|w|$ it is easy to see, using properties (1) and (2) above, that the reduction sequence of Mw to its normal form has length polynomial in $|w|$. $\dashv$

2.3. Solitary inferences and poly-time

Proposition 3 shows that restricting induction to positive formulas can be traded for a prohibition of assumption multiplicity. The class of provable functions is poly-time in either case. While this result is potentially beneficial in some cases, it seems that multiple invocation of assumptions is, in fact, used and needed in actual proofs far more frequently than induction over data-complex formulas. Fortunately, we can combine the advantages of both approaches. Call a formula data-complex if the contracted form of $\kappa\varphi$ contains $\rightarrow\!\!\!\!\!\!\rightarrow$. Call a derivation $\mathcal{D}$ *solitary* if all *data-complex* assumption classes are solitary.

THEOREM 4 *A function f is poly-time iff it is provable by a solitary and predicative derivation $\mathcal{D}$.*

Note that induction is permitted here over all formulas. If induction is restricted to positive formulas, then $\mathcal{D}$, which w.l.o.g. can be assumed normal, has no data-complex assumption classes, so Theorem 2 is a special case of Theorem 4.

Proof Outline. The proof of Proposition 3 shows that every poly-time function is provable as required.

For the converse, consider a derivation $\mathcal{D}$ as above. The term $\kappa\mathcal{D}$, which by Theorem 1(1) defines f, has only predicative instances of recurrence, and no higher-type variable is multiply-closed by a λ-abstraction in $\kappa\mathcal{D}$. In [15, Lemma 3.12] we showed that such terms define poly-time functions. $\dashv$

2.4. Origins of solitary deductions

It has been known for long that allowing resources to be invoked only once is related to poly-time computation. For instance, linear logic leads to poly-time [6, 5], second-order existential database queries are poly-time if the matrix is Horn [12, 8], monotone inductive definitions (where each object is inserted only once) define exactly the poly-time queries over finite structures, ramified recurrence with parameters does not lead out of poly-time if only one parameter is used [1], Turing machines operating in poly-space accept exactly the poly-time languages if non-blank tape-cells cannot be reused, etc.

Continuing in this vein, Martin Hofmann [9, 10, 11] has developed a linear-type ramified functional calculus that defines exactly the poly-time non-size-increasing functions, even if recurrence is used at all finite types. This has been further refined in [3]. Independently we showed that allowing abstracted higher order functions to be used only once in λ-recurrence terms, yield exactly poly-time [15].

Proof theoretic characterizations of poly-time that build on linearity have also appeared recently, among others in [2, 22]. The main advantage of our present result is that it does not require an overlay of syntactic machinery on the formulas; the proofs we consider are all proofs in intrinsic theories (whose syntax is very simple), and the structural properties that they satisfy can be automatically checked. This yields a transparent machinery for certifying program feasibility.

Since poly-time has rather simple proof-theoretic characterizations (e.g. [16] above), the main motivation of restricted-multiplicity conditions is the attempt to permit induction for all formulas, thereby providing the user of the formalism (human or automated) with a larger arsenal. Consequently, it is self-defeating to abandon in the process other methods, in particular when these are important and natural. For instance, taken in isolation, restricted multiplicity disallows a direct and simple proof of the squaring function! Indeed, one would wish to combine the advantages of various approaches, rather than piling up the hurdles to using them.

There is a trade-off, of course, in our avoiding the explicit use of linear and other resource-control operators. Our combinatorial conditions may be viewed as corresponding to the use of such operators at the outer level of reasoning, whereas more explicit resource-control operators, as in [10], might be used to convey more subtle interactions between parts of proofs. Examples illustrating such gains are yet to be developed, however.

3. POLY-SPACE

3.1. Weakly-solitary derivations

Returning to our example above of a derivation for the functions 2^x+y, we can further ask: is the culprit for exponential growth-rate the very duplicate use of the assumption, or only the particular setting where one use is applied to another, across an instance of implication elimination? In other words, what would happen if we allow duplicate uses of data-complex working assumptions, as long as they are not ancestors of distinct premises of implication elimination? Interestingly, the provable functions are then precisely the functions computable in polynomial space.

Call an assumption-class in a natural deduction $\mathcal{D}$ *weakly-solitary* if no two of its members are ancestors of distinct premises of an instance of implication elimination. A derivation $\mathcal{D}$ is *weakly-solitary* if every data-complex assumption-class in $\mathcal{D}$ is weakly-solitary.

We show below that a function $f : \mathbb{W}^* \rightarrow \{0,1\}$ is provable by predicative and weakly solitary derivations iff it is computable in polynomial space. Recall that a function $g : \mathbb{W}^* \rightarrow \mathbb{W}$ is computable by a Turing machine in polynomial space iff the associated bit-function

$$g'(\vec{x}, y) =_{\mathrm{df}} \text{the bit of } g(\vec{x}) \text{ at address } y$$

is computable in PSpace.[7] However, our proof does not apply directly to functions g as above, since it relies on the characterization of PSpace as alternating PTime [4]. With this in mind, we will prove the following.

THEOREM 5 *A function* $f : \mathbb{W}^* \rightarrow \{0,1\}$ *is provable in* $\mathbf{IT}(\mathbb{W})$ *by a predicative and weakly-solitary derivation iff it is in PSpace.*

3.2. From PSpace to provability

PROPOSITION 6 *Every function* $f : \mathbb{W}^* \rightarrow \{0,1\}$ *computable in polynomial space is provable in* $\mathbf{IT}(\mathbb{W})$ *by a predicative and weakly-solitary derivation.*

Proof Outline. The proof uses a proof-theoretic analog of the technique of [18, 19]. By [4] a boolean-valued function computable in PSpace is computable in polynomial time by an alternating Turing machine, which w.l.o.g. has branching degree 2. The latter is computable by composing a polynomial function (generating the computation clock) with a function defined by parameterized recurrence of the form:

$$\begin{aligned} f(\varepsilon, \vec{x}) &= g_\epsilon(\vec{x}) \\ f(\mathbf{c}t, \vec{x}) &= g_s(\vec{x}, f(t, \vec{h}_0(\vec{x})), f(t, \vec{h}_1(\vec{x}))) \end{aligned}$$

The intent is that $f(t, \vec{x})$ is the acceptance status returned by the given alternating machine, when in configuration $\vec{x}$, and provided with $|t|$ computation steps along each branch. The functions h_i return the two subsequent configuration, and are defined explicitly without use of recurrence (other than definition by cases). The function g_s returns the conjunction or disjunction of its last two arguments, depending on the state whose code is part of $\vec{x}$.

It is easy to see that the equational program described above is provable, using induction for the formula

$$\varphi[t] \equiv_{\mathrm{df}} \forall \vec{x}. \mathsf{W}(\vec{x}) \rightarrow \mathsf{W}(\mathsf{f}(t, \vec{x}))$$

The formula is used twice in the induction step, with $\vec{x}$ instantiated once to $\vec{h}_0(\vec{x})$ and once to $\vec{h}_1(\vec{x})$. The two instances are combined using only basic operations, without use of implication elimination or induction. $\dashv$

[7]More precisely, g' has co-domain $\{0, 1, \emptyset\}$, and returns $\emptyset$ if the address y exceeds the length of $f'(\vec{x})$.

3.3. From Provability to PSpace

We complete the proof of Theorem 5 by showing that every function f provable by a predicative and weakly-solitary derivation is in PSpace.

Suppose $\mathcal{D}$ is a predicative and weakly-solitary derivation. The term $M \equiv \kappa\mathcal{D}$ has the following properties: (1) No recurrence argument has a free variable bound in M; and (2) M is ***weakly-solitary*** in the following sense: If EF is a subterm of M, then no variable of higher type occurs free in both E and F. Call a $\lambda_1(\mathbb{W})$-term $\lambda\vec{x}.M[\vec{x}]$ ***input-driven*** if the recurrence arguments in M are all variables out of the list $\vec{x}$. Note that this is a stronger condition than (1) above.

LEMMA 7 *Every function represented in $\lambda_1(\mathbb{W})$ by a predicative and weakly-solitary term is the composition of functions represented in $\lambda_1(\mathbb{W})$ by input-driven and weakly-solitary terms.*

See [15, Lemma 2.2] for a proof. The proof of Theorem 5 is now concluded by the following.

LEMMA 8 *If a function f is representable in $\lambda_1(\mathbb{W})$ by a weakly-solitary and predicative term, then it is computable in polynomial space.*

Proof Outline. By Lemma 7 it suffices to consider the case where f is representable by a weakly-solitary and input-driven term $\lambda\vec{x}.F$. We may further assume w.l.o.g. that the recurrence arguments in F are *distinct* variables out of the list $\vec{x}$. Let $y_1, \ldots, y_q$ be the x_i's used for higher order recurrence, and $z_1, \ldots, z_k$ the x_i's used for recurrence in $\rightarrow$-free types. Given $\mathbf{y}_1, \ldots, \mathbf{y}_q, \mathbf{z}_1, \ldots, \mathbf{z}_k \in \mathbb{W}$, consider the term $F^* = \{\vec{\mathbf{y}}, \vec{\mathbf{z}}/\vec{y}, \vec{z}\}F$. By the condition above we can independently unfold each higher-type recurrence in F^*, finally yielding some term M. Clearly, M is of polynomial height (and exponential size).

Let m the maximal number of occurrences of higher-type variables that are bound by a λ-abstraction in F. Consider the symbolic parse-tree T_M of M, with the root at the bottom. Each node in $T(M)$ has below it the junctures of recurrence-unfolding on F^*. Each such juncture corresponds to one choice out of (at most) m positions, for a λ-abstracted variable. Thus, to each node there corresponds a "reduction address" in $\{0, \ldots, m-1\}^h$, where $h \leq$ the height of M. Call a node N in T_M *relevant* to node N' if the reduction address of N is a subsequence (not necessarily strict) of the reduction address of N'. It can then be seen that: (a) Each node has only polynomially many relevant nodes relevant to it; and (b) The behavior of each node under reductions can be affected only by nodes relevant to it (here the definition of weakly-solitary terms is crucial). Consider now a reduction sequence on M that eliminates higher-type redexes. Since M is itself weakly-solitary, no node of T_M can be duplicated on the same branch of any redex term. It follows that for each term M' along the reduction sequence, the parse-tree $T(M')$ is the "horizontal union" of (perhaps exponentially many) subtrees of polynomial size and height: each such subtree corresponds to a reduction address in M.

Consequently, the entire reduction sequence of M can be computed in polynomial-space, leading to an input-driven and predicative term whose only redexes are for positive-type reductions. This can now be normalized in space polynomial in the height of the term, which is itself polynomial in the size of the input. The final normal form is the value of F^*, i.e. the value of the function f for input $\mathbf{y}_1, \ldots, \mathbf{y}_q, \mathbf{z}_1, \ldots, \mathbf{z}_k \in \mathbb{W}$. $\dashv$

References

[1] A. Beckmann and A. Weiermann. A term rewriting characterization of the polytime functions and related complexity classes. *Archive for Mathematical Logic*, 36:11–30, 1996.

[2] Stephen Bellantoni and Martin Hofmann. A new feasible arithmetic. *Journal for Symbolic Logic*, 2001.

[3] Stephen J. Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104 (1-3):17–30, 2000.

[4] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[5] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143, 1998.

[6] Jean-Yves Girard, Andre Scedrov, and Philip Scott. Bounded linear logic: A modular approach to polynomial time computability. *Theoretical Computer Science*, 97:1–66, 1992.

[7] Kurt Gödel. Über eine bisher noch nicht benutzte erweiterung des finiten standpunktes. *Dialectica*, 12:280–287, 1958.

[8] E. Grädel. Capturing Complexity Classes by Fragments of Second Order Logic. *Theoretical Computer Science*, 101:35–57, 1992.

[9] Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Proceedings of CSL'97*, pages 275–294. Springer-Verlag LNCS 1414, 1998.

[10] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of LICS'99*, pages 464–473. IEEE Computer Society, 1999.

[11] Martin Hofmann. Safe recursion with higher types and bck-algebra. *Annals of Pure and Applied Logic*, 104 (1-3):113–166, 2000.

[12] Daniel Leivant. Descriptive characterizations of computational complexity. In *Second Annual Conference on Structure in Complexity Theory*, pages 203–217, Washington, 1987. IEEE Computer Society Press. Revised in *Journal of Computer and System Sciences*, 39:51–83, 1989.

[13] Daniel Leivant. Contracting proofs to programs. In P. Odifreddi, editor, *Logic and Computer Science*, pages 279–327. Academic Press, London, 1990.

[14] Daniel Leivant. Intrinsic theories and computational complexity. In D. Leivant, editor, *Logic and Computational Complexity*, LNCS, pages 177–194, Berlin, 1995. Springer-Verlag.

[15] Daniel Leivant. Applicative control and computational complexity. In J. Flum and M. Rodriguez-Artalejo, editors, *Computer Science Logic (Proceedings of the*

Thirteenth CSL Conference, pages 82–95, Berlin, 1999. Springer Verlag (LNCS 1683).

[16] Daniel Leivant. Termination proofs and complexity certification. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software (TACS 2001)*, Springer LNCS 2215, pages 183–200, 2001.

[17] Daniel Leivant. Intrinsic reasoning about functional programs I: first order theories. *Annals of Pure and Applied Logic*, 114:117–153, 2002.

[18] Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity IV: Predicative functionals and poly-space. *Information and Computation.* To appear.

[19] Daniel Leivant and Jean-Yves Marion. Predicative functional recurrence and poly-space. In M.Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development*, LNCS 1214, pages 369–380, Berlin, 1997. Springer-Verlag.

[20] Edward Nelson. *Predicative Arithmetic.* Princeton University Press, Princeton, 1986.

[21] Kurt Schütte. *Proof Theory.* Springer-Verlag, Berlin, 1977.

[22] Helmut Schwichtenberg. An arithmetic for polynomial-time computation. Submitted for publication, 2002.

$\mathcal{D}$	$\kappa\mathcal{D}$
$\overset{(\ell)}{\psi}$ (labeled assumption)	$x_\ell^{\kappa\psi}$ (ℓ-th variable of type $\kappa\psi$)
$\dfrac{\overset{\mathcal{D}_0}{\varphi_0} \quad \overset{\mathcal{D}_1}{\varphi_1}}{\varphi_0 \wedge \varphi_1}$	$\langle \kappa\mathcal{D}_0, \kappa\mathcal{D}_1 \rangle$
$\dfrac{\overset{\mathcal{D}_0}{\varphi_0 \wedge \varphi_1}}{\varphi_i}$	$\pi_i \kappa\mathcal{D}_0$
$\dfrac{\overset{\overset{\overset{(\ell)}{\psi}}{\mathcal{D}_0}}{\varphi}}{\varphi \to \psi}\,(\ell)$	$\lambda x_\ell^{\kappa\psi}.\,\kappa\mathcal{D}_0$
$\dfrac{\overset{\mathcal{D}_0}{\varphi \to \psi} \quad \overset{\mathcal{D}_0}{\varphi}}{\psi}$	$(\kappa\mathcal{D}_0)(\kappa\mathcal{D}_1)$
$\dfrac{\overset{\mathcal{D}_0}{\varphi[z]}}{\forall x\, \varphi[x]}$	$\kappa\mathcal{D}_0$
$\dfrac{\overset{\mathcal{D}_0}{\forall x\, \varphi[x]}}{\varphi[\mathsf{t}]}$	$\kappa\mathcal{D}_0$
$\mathsf{t} = \mathsf{t}$	$*$
$\dfrac{\overset{\mathcal{D}_0}{\mathsf{t} = \mathsf{t}'} \quad \overset{\mathcal{D}_1}{\varphi[\mathsf{t}]}}{\varphi[\mathsf{t}']}$	$\kappa\mathcal{D}_1$
$\mathsf{W}(\varepsilon)$	0
$\dfrac{\overset{\mathcal{D}_0}{\mathsf{W}(\mathsf{t})}}{\mathsf{W}(\mathsf{it})} \quad (i = 0, 1)$	$\mathsf{i}\kappa\mathcal{D}_0$
$\dfrac{\overset{\mathcal{D}_0}{\mathsf{W}(\mathsf{it})}}{\mathsf{W}(\mathsf{t})} \quad (i = 0, 1)$	$\mathsf{p}\kappa\mathcal{D}_0$
$\dfrac{\overset{\mathcal{P}}{\mathsf{W}(\mathsf{t})} \quad \overset{\mathcal{D}_\epsilon}{\varphi[\varepsilon]} \quad \overset{\mathcal{D}_0}{\varphi[z] \to \varphi[\mathbf{0}z]} \quad \overset{\mathcal{D}_1}{\varphi[z] \to \varphi[\mathbf{1}z]}}{\varphi[\mathsf{t}]}$	$\mathsf{R}_{\kappa\varphi}(\kappa\mathcal{D}_\epsilon)(\kappa\mathcal{D}_0)(\kappa\mathcal{D}_1)(\kappa\mathcal{P})$