ELSEVIER

Theoretical Computer Science 139 (1995) 27–67

Theoretical Computer Science

Fundamental Study

On the logic of UNITY☆

Peter Päppinghaus*

SIEMENS AG, Corporate Research and Development, Basic Technologies, Software and Engineering, Otto-Hahn-Ring 6, D-81739 München, Germany

Received December 1992; revised February 1994
Communicated by M. Wirsing

Abstract

The UNITY approach to specification, design, and verification of parallel programs expounded by Chandy and Misra (1988) contains a programming notation, and a programming logic to express and prove properties of UNITY programs. For progress properties there is a basic notion **leads-to**, which comes in two versions. One version – here called **leads-to** – explicitly expresses progress of fair execution sequences of a UNITY program. Another prima facie stronger version – here called $\mapsto$ – is defined by infinitary closure conditions from the more elementary notion **ensures**. This version is used in the actual proofs of program properties given in (Chandy and Misra, 1988). In this paper these notions are investigated from a foundational point of view.

A principle of transfinite induction for $\mapsto$ is introduced, and it is proved that every true **leads-to** proposition can be obtained by application of one single instance of this principle. Semantic completeness of $\mapsto$ w.r.t. its intended semantics **leads-to** is a corollary.

Aspects of the complexity and computational power of UNITY are analyzed. The halting problem for fair executions of UNITY programs is shown to be Π_1^1-complete, and it is proved that a nondeterministic numerical function is computable by a UNITY program if and only if its graph is Σ_1^0 and its halting set is a Π_1^1-subset of its domain. In order to retain completeness of the principle of transfinite induction for $\mapsto$, arbitrarily large recursive ordinals are needed as heights of well-founded relations.

A formalized finitary proof system for $\mapsto$ is introduced and proved syntactically correct and complete relative to the set of true Π_1^1-sentences of a second-order assertion language. More precisely, the well-founded relation itself and the auxiliary assertions used can be expressed in a first-order language. It is only the hypothesis of well-foundedness, which has to be expressed by a Π_1^1 formula. In view of the Π_1^1-completeness phenomenon, this result is best possible.

☆This work was partially supported by the BMFT project KORSO grant no. 01 IS 203 L8.
* E-mail: pae@zfe.siemens.de.

SSDI 0304-3975(94)00043-I

Contents

1. Introduction

UNITY (Unbounded Nondeterministic Iterative Transformations) is an approach to specification, design, and verification of parallel programs. It originates from the work of Chandy and Misra, and since publication of their book [5] has attracted a lot of attention. Besides a philosophy and a methodology, UNITY consists of two parts: a *programming notation* (simple programming language), and a *programming logic* to express and prove properties of programs.

A UNITY program essentially is a finite, nonempty set of multiple conditional assignments, which are assumed to be deterministic, terminating, and always applicable. The *operational semantics* of a UNITY program is given via a finite transition system, whose transitions are total and deterministic functions over an underlying state space. An *execution sequence* of a UNITY program is an infinite sequence of states s.t. each successor state is obtained from its predecessor by a transition. There is no notion of *final state*. Instead an execution is considered finished when the execution sequence has reached a *fixed point*, i.e., a state which is not changed by any transition. Apart from the conditions in the assignments, there is no control construct whatsoever. The control rests solely in an overall fairness assumption. An execution sequence is called *fair* if every transition is executed infinitely often.

Comparison: UNITY is in some respects simpler than other nondeterministic languages studied earlier in the literature. Compared to Dijkstra's language of *guarded commands* (GC), which is the basis of Francez' study of fairness [8], there is no sequential composition and no nesting of selection and repetition (cf. [8, pp. 7ff.]). A UNITY program can be considered to be a GC program consisting of a single repetition loop using only simultaneous assignments in its constituent guarded commands. There is no notion of *enabledness* as in GC, and also there is no notion of *failure state*. The replacement of the notion of final state by the notion of fixed point is a minor difference. The lack of a notion of enabledness in UNITY is relevant only for

fairness considerations and corresponds to restricting attention to *unconditional fairness* in the sense of [8, p. 25].

In reasoning about parallel programs, the significant properties can usually be classified as either *safety* or *progress* (*liveness*) properties. The central notions of the programming logic of UNITY are unless for safety and leads-to for progress properties. leads-to explicitly expresses progress of fair execution sequences of a UNITY program. For sets of states **P** and **Q**, **P** leads-to **Q** means that every fair execution sequence arriving at a state in **P** will eventually arrive at a state in **Q**.

In designing a practical programming logic, an important discovery of Chandy and Misra is the observation that it is better not to reason directly with this notion of leads-to, but rather with another notion denoted by $\mapsto$.[1] $\mapsto$ is defined inductively (by infinitary closure conditions) from a more elementary notion called ensures. leads-to satisfies the closure conditions defining $\mapsto$, in other words $\mapsto$ is a prima facie stronger notion than leads-to. This fact is alluded to as *semantic correctness* of $\mapsto$ w.r.t. leads-to. Arguing with ensures has mainly two advantages. Firstly it is a local notion stating that a single transition ensures progress of a computation in a suitable sense, and secondly it is well behaved w.r.t. composition of programs.

Aim. In an ongoing project at the Corporate Research and Development Laboratories of the SIEMENS AG, it is intended to apply the UNITY approach to the design and specification of concurrent systems, and to implement software supporting the interactive verification of parallel programs. The aim of this paper is to carry out some foundational work useful to clarify which formalism should be implemented. We focus attention on investigation of the notions leads-to and $\mapsto$.

As yet there is no 'logic of UNITY' in the sense of a finitary formal system with syntax and semantics, and a clarified relation between them. The treatment of the logic of UNITY in [5] is semi-formal, the emphasis being on case studies to demonstrate the viability of the UNITY approach. Of course, formulas are used to describe sets of states, and rules about $\mapsto$ are proved and used. However, if one wants to view these as constituting a proof system, this is an infinitary system and not a formal one, and as such it is not implementable. In our opinion the proofs in [5] should not be taken as proving rules of a proof system, but rather as establishing closure conditions of $\mapsto$, where $\mapsto$ is viewed as a semantic relation on subsets of the state space associated with a UNITY program.

In this paper we develop a *formalized logic of UNITY* to an extent satisfactory from a theoretical point of view. More work has to be done to obtain a formalized logic of UNITY which is satisfactory also in a practical sense. This is discussed in Section 6.

Contents and Results. Our formalized logic of UNITY is presented at the end of the paper in Section 5. A formal language is introduced for expressing program properties of a fixed UNITY program $\mathscr{T}$. This language extends a many-sorted, first-order

[1] In [5], our notion leads-to is used only in informal explanations of $\mapsto$ (cf. [5, p. 52])], there is no separate notation for it. Here, however, we use different notations to distinguish more carefully between these notions.

language of predicate logic by new logical operators called *program quantifiers*, $\mapsto_{\mathscr{F}}$ among others. The semantics of these program quantifiers is defined using corresponding relations on subsets of the state space. These relations are refinements of the notions of Chandy and Misra similar to those proposed by Sanders in her paper [24] clarifying the role of the so-called *substitution axiom*. A formal (finitary) proof system for $\mapsto_{\mathscr{F}}$ formulas is defined and shown to be correct and relatively complete in a suitable sense. As in Hoare logic, completeness in an absolute sense cannot be expected for a finitary proof system.

The bulk of this paper is concerned with investigating the UNITY notions on the semantic side so as to arrive at the desired completeness result and to clarify in which sense relative completeness can be expected. For this we draw on work in the theory of nondeterministic programming languages done in the last two decades. Comprehensive sources containing more references are [3] for countable nondeterminism and [8] for fairness in finite nondeterminism.

In Section 2 the basic notions are defined. In particular the *operational semantics* of a UNITY program is given via a suitable notion of *transition system*. As long as we work solely on the semantic side, our notions and results apply to such transition systems in general, not only to those induced by a UNITY program.

The intended semantics of the program quantifier $\mapsto_{\mathscr{F}}$ is a refined version of leads-to, a notion explicitly expressing progress of fair execution sequences of the program $\mathscr{F}$. The semantic relation $\mapsto$, defined by an infinitary inductive definition, is much closer to a proof rule and is as mentioned contained in leads-to (semantic correctness). As the crucial step towards a syntactically complete proof rule for $\mapsto_{\mathscr{F}}$ we prove *semantic completeness* of $\mapsto$ w.r.t. leads-to (i.e. containment of leads-to in $\mapsto$). In order to tackle this problem, we introduce in Section 3 a *principle of transfinite induction* for $\mapsto$ and prove semantic completeness of this principle w.r.t. leads-to. This result implies semantic completeness of $\mapsto$ w.r.t. leads-to[2] and yields a *normal form* for $\mapsto$: it is shown that every true leads-to proposition can be obtained by application of a single instance of the transfinite induction principle, and this instance is such that only ensures relations are used in the premisses. Our proof is fully explicit and its formalization immediately yields syntactic completeness of a corresponding formalized principle of transfinite induction for $\mapsto_{\mathscr{F}}$. With one exception, only first-order formulas and program quantifiers on top of first-order formulas occur in this proof system for $\mapsto_{\mathscr{F}}$. The exception just mentioned is the Π^1_1-formula expressing *well-foundedness* of a suitable first-order definable partial order occurring in the principle of transfinite induction. Our proof system is shown to be syntactically correct and complete relative to the set of Π^1_1-formulas valid in a so-called *arithmetical structure*.[3]

[2] Meanwhile other direct proofs of semantic completeness of $\mapsto$ w.r.t. leads-to have appeared in the literature (see the discussion of related work in Section 6).

[3] A notion similar to ours is familiar from the literature (e.g. [1, p. 442]). In particular it requires the first-order fragment of the assertion language to be an arithmetical language of sufficient expressive power.

A closer look at the completeness proof gives even more information on definability. If a valid $\mapsto_{\mathcal{F}}$ formula is built by Δ_0^0-formulas, then the auxiliary well-founded partial order occurring in its derivation is Δ_0^0-definable, and the only other auxiliary formula occurring in the derivation is Σ_1^0.

Our proof system gives a nice separation of the Π_1^1-aspect – isolated in the property of well-foundedness of auxiliary partial orders – from the remaining elementary aspects. The assertion language used is a 'minimal' extension of first-order predicate logic.

In order to show that the use of a Π_1^1-oracle is indispensable for a relatively complete finitary proof system for $\mapsto_{\mathcal{F}}$, we study in Section 4 the recursion theoretic complexity of leads-to. It turns out that the *fairness condition* gives rise to complexity 'Π_1^1-complete' and to the necessity of using arbitrarily large recursive ordinals as ordinal heights of well-founded partial orders in the transfinite induction principle for $\mapsto$. This is no surprise, since analogous results are well known for other nondeterministic languages. What our results show is that UNITY has essentially the same expressive and computational power as earlier more complicated nondeterministic languages like, for example, GC.

The recursion theoretic analysis of Section 4 applies to recursive transition systems in general, not only to 'UNITY-induced' ones. In detail the results are the following. In Section 4.2 the complexity of the halting problem for recursive transition systems is studied. By the *halting problem* for a particular transition system we mean the set of all states s.t. *every* fair execution sequence reaches a fixed point.[4] Just as in the deterministic case this notion is generalized to the notion of *general halting problem* for a class of transition systems. The general halting problem for recursive transition systems is shown to be Π_1^1, and a specific UNITY program is exhibited whose halting problem is Π_1^1-complete. This improves a result of Pachl in [21, Section 4] who shows the general halting problem for UNITY programs to be of complexity at least Π_1^0 by means of a reduction to the complement of *Post's Correspondence Problem.*

Section 4.3 is concerned with the *computational power* of UNITY. To measure computational power, attention is restricted to UNITY programs computing over the natural numbers. Since execution of UNITY programs is nondeterministic, such programs are viewed as computing *nondeterministic* numerical functions. As Chandra has pointed out in [4], it is not sufficient to look at the complexity of the graphs of computable nondeterministic functions. In the nondeterministic case, the function graph does not characterize the I/O-behavior of a program adequately. One further has to distinguish between inputs for which *some* and those for which *every* fair execution sequence reaches a fixed point (i.e. whether, so to speak, 'undefined' is a 'value' of the function or not). Whereas the former set of input values, called the *domain* of the function, can be defined from its graph, the latter subset of the domain,

[4] The weaker notion of *some* fair execution sequence reaching a fixed point is less significant and easily seen to be of complexity Σ_1^0.

called the *halting set* of the function, plays an independent role. We show that the result of Chandra for computations of nondeterministic register machines [4, Theorem 2, p. 129] also holds for UNITY programs: a nondeterministic numerical function is computable by a UNITY program if and only if its graph is Σ_1^0, and its halting set is a Π_1^1-subset of its domain. This improves a result of Gerth and Pnueli in [9], who study UNITY by comparing it with a closely related class of programs in the language of guarded commands, called SLP (*single location programs*). They observe that the class of SLP-programs is universal in the sense that every partial recursive function can be computed by an SLP-program. Applying our result to deterministic functions, we obtain that a *deterministic* partial function over the natural numbers can be computed by a UNITY program if and only if it is partial recursive.

In Section 4.4 the complexity of the constructions in the proof of the semantic completeness theorem for $\mapsto$ is analyzed. It is proven that for recursive sets of states, the well-founded partial order used in the transfinite induction principle for $\mapsto$ can be chosen to be recursive and an auxiliary family of sets of states to be recursively enumerable. This implies that the ordinal height of this well-founded partial order is a recursive ordinal. On the other hand, for any fixed recursive ordinal α, the restriction of the transfinite induction principle to well-founded relations of ordinal height less than α is not complete any more w.r.t. $\mapsto$ relations between recursive sets of states. This is shown by exhibiting a suitable family of $\mapsto$ relations true in a specific UNITY program.

2. Basic notions

UNITY Programs. To concentrate on the essential issues, we discuss a simplified version of UNITY leaving out derivable constructs. A UNITY program consists of three sections: the declare-section, the initially-section, and the assign-section.

The *assign-section* is a finite, nonempty set of *multiple conditional assignments* of the following form:

$$\begin{aligned} \boldsymbol{x} \coloneqq\ & \boldsymbol{t}_0 \quad \text{if } e_0 \\ & \vdots \\ & \sim \boldsymbol{t}_n \quad \text{if } e_n. \end{aligned}$$

$\boldsymbol{x}$ denotes a finite vector of program variables, $\boldsymbol{t}_0, \ldots, \boldsymbol{t}_n$ denote corresponding vectors of *terms*, and $e_0, \ldots, e_n$ denote *boolean expressions*. The variables, terms, and boolean expressions belong to a fixed many-sorted language interpreted in a suitable fixed many-sorted algebra. The program is said to *compute over* this algebra. The function symbols of the language are required to denote *total, deterministic* functions. The assignments are subject to the restriction that the case distinction introduces no ambiguity, i.e., an assignment is allowed only provided the values of the assigning terms are the same, if more than one condition holds.

The *declare-section* contains declarations of the sorts (alias datatypes) of all variables occurring in the program. For short, these variables are called the *program variables.*

The *initially-section* initializes program variables (not necessarily all of them) by equations taking the same form as those used in the assignment statements.

Operational semantics. To define the operational semantics of a UNITY program, the cartesian product of the *carriers* of its program variables[5] is associated with it as *state space* **S**. Each assignment statement is interpreted as to induce a *transition*, which is a *total* and *deterministic* function from **S** to **S**. A UNITY assignment is considered to be always applicable. If none of the assignment's conditions holds, it simply maps a state to itself. There is no notion of *enablednesss* as in Dijkstra's guarded commands language, and also there is no notion of *failure state.*

An *execution sequence* of a UNITY program is an infinite sequence of states s.t. each successor state is obtained from its predecessor by a transition. A finite contiguous part of an execution sequence is called an *execution segment.* There is no notion of *final state.* Instead an execution is considered finished, when it has reached a *fixed point*, i.e., a state which is not changed by any transition. Apart from the conditions in the assignments, there is no control construct whatsoever. The control rests solely in an *overall fairness assumption.* An execution sequence is called *fair* if every assignment of the program is executed infinitely often, and it is called *legal* if it is fair and its initial state satisfies the equations in the initially-section of the program.

According to the explanations given above a UNITY program induces a transition system in the following sense.

Definition 2.1 (*Transition system*). Let **S** be a nonempty set, called the set of states or *state space.* $\mathscr{T}$ is called a *transition system over* **S** iff $\mathscr{T}$ is a finite, nonempty set and for every $\tau\in\mathscr{T}$, $\tau:\mathbf{S}\to\mathbf{S}$ is a total function.

To a large extent, our results will be shown for arbitrary transition systems in the sense of this definition, regardless of whether they are induced by UNITY programs. The notions of execution sequence, execution segment, fixed point and fairness are transferred to transition systems in the obvious way. We have not required a set of initial states to be part of a transition system, since it will not play an essential role in our treatment.

Program properties. As mentioned in Section 1, the notions studied in this paper are a refinement of the original definitions of the basic program properties in [5]. This refinement is adopted for the following reasons.

The book of Chandy and Misra contains an ambiguity as to whether the quantifiers over 'states' in their notions range over the state space **S** or over the subspace $\overline{\mathbf{Init}}$ of all *reachable states.*[6] The *Union Theorem* for program composition (see [5, p. 155]) taken

[5] By the carrier of a program variable we mean the set of data elements of its datatype.

[6] See, for example, the explanations of $\{\mathbf{P}\}s\{\mathbf{Q}\}$ in [5, p. 42] versus [5, p. 46].

literally is correct only if quantification over **S** is assumed. The practice of proving program invariants and tacitly using them in proofs of further program properties sanctioned by the *Substitution Axiom* of [5, p. 49], however, is correct only if quantification over $\overline{\mathbf{Init}}$ is assumed. There has been a debate over this issue between Misra and Sanders (see [18, 24]). Sanders suggests to introduce refinements of the notions of UNITY logic by relativizing them to a program invariant. Quantifiers are then taken to range over **S**. Misra favors the reachable-state-version of the notions. In effect, he argues that a strongest invariant actually used can be retrieved from a given proof, even if there is no book-keeping device such as Sanders' relativized notions. Of course one is ultimately interested in reachable states, but observe that Sanders' relativized notions are more general. The reachable-state-version can be expressed in terms of these notions simply by taking $\overline{\mathbf{Init}}$ as the program invariant, to which one relativizes, and the state-space-version can be expressed by relativizing to **S**. Moreover, Sanders' definitions allow the Substitution Axiom to be eliminated in favor of corresponding theorems.

We follow Sanders' approach and define and study relativized notions. In contrast to Sanders, however, the sets of states to which we relativize are just required to be stable, not necessarily program invariants.

In order to emphasize our distinction between syntax and semantics, we use set-theoretic notation when dealing with transition systems on the semantic side.

In the rest of this and in the following section, we tacitly refer to a fixed transition system $\mathscr{T}$, which is assumed to be given. Throughout we use **S** to denote the state space of a transition system. Uppercase boldface letters range over subsets of this state space, $\sigma, \sigma', \sigma_1, \ldots$ range over states, and $\tau, \tau', \tau_1, \ldots$ range over transitions.

The basic safety properties of UNITY logic are defined as follows.

Definition 2.2 (*Safety properties*). (1) $\mathsf{stable}\,\mathbf{P} \overset{\text{def}}{\Leftrightarrow} \forall \tau \in \mathscr{T}\colon \tau(\mathbf{P}) \subseteq \mathbf{P}$ and
(2) $\mathbf{P}\,\mathsf{unless}_{\mathbf{I}}\,\mathbf{Q} \overset{\text{def}}{\Leftrightarrow} \mathsf{stable}\,\mathbf{I} \wedge \forall \tau \in \mathscr{T}\colon \tau((\mathbf{I} \cap \mathbf{P}) \setminus \mathbf{Q}) \subseteq \mathbf{P} \cup \mathbf{Q}$.

Remarks. (1) There is no point in defining a relativized version of stable. $\mathsf{stable}_{\mathbf{I}}\,\mathbf{P}$ would be equivalent to $\mathbf{P}\,\mathsf{unless}_{\mathbf{I}}\,\emptyset$, which in turn is equivalent to $\mathsf{stable}\,\mathbf{I} \cap \mathbf{P}$.

(2) If a transition system is induced by a UNITY program, then the notion of a program invariant can be defined as follows:

$$\mathsf{invariant}\,\mathbf{P} \overset{\text{def}}{\Leftrightarrow} \mathsf{stable}\,\mathbf{P} \wedge \mathbf{P} \supseteq \mathbf{Init},$$

where **Init** is defined to be the set of all states satisfying the initial conditions of the program. In this paper we will, however, not need this notion. To prove something about the relativized properties, it is useful that **I** be stable. It turns out that **I** being an invariant is not essential.

Two basic progress properties are defined for UNITY logic, the first is defined operationally, and the second inductively.

Definition 2.3 (*Operational progress property*). $\mathbf{P}$ leads-to$_{\mathbf{I}}$ $\mathbf{Q} \overset{\text{def}}{\Leftrightarrow}$ stable $\mathbf{I}$ and every fair execution sequence starting in a state in $\mathbf{I} \cap \mathbf{P}$ will in this or a later step arrive at a state in $\mathbf{Q}$.

For conciseness, certain self-explanatory phrases like 'an execution sequence from $\mathbf{P}$ reaches $\mathbf{Q}$', 'an execution sequence from $\mathbf{P}$ avoids $\mathbf{Q}$', etc., will henceforth be used without defining them formally.

Remarks. (1) From the informal explanations in Section 1, one would expect a slightly different definition, namely: $\mathbf{P}$ leads-to$_{\mathbf{I}}$ $\mathbf{Q} \Leftrightarrow$ every fair execution sequence from $\mathbf{I}$ reaching $\mathbf{P}$ will at that or a later step arrive at a state in $\mathbf{Q}$. For a stable set $\mathbf{I}$, this is equivalent to our definition by virtue of the following observation.

(2) An execution sequence from a stable set $\mathbf{I}$ cannot leave $\mathbf{I}$. So instead of looking at execution sequences from $\mathbf{I}$ reaching $\mathbf{P}$, it suffices to consider execution sequences from $\mathbf{I} \cap \mathbf{P}$. Every execution sequence from $\mathbf{I}$ reaching $\mathbf{P}$ can be transformed to an execution sequence from $\mathbf{I} \cap \mathbf{P}$ by cutting off an initial segment.

Definition 2.4 (*Inductive progress property*). (1) $\mathbf{P}$ ensures$_{\mathbf{I}}$ $\mathbf{Q} \overset{\text{def}}{\Leftrightarrow} \mathbf{P}$ unless$_{\mathbf{I}}$ $\mathbf{Q} \wedge \exists \tau \in \mathscr{T}\colon \tau((\mathbf{I} \cap \mathbf{P}) \setminus \mathbf{Q}) \subseteq \mathbf{Q}$.

(2) $\mapsto$ is defined to be the least relation satisfying the following closure properties:

(1) $\mathbf{P}$ ensures$_{\mathbf{I}}$ $\mathbf{Q} \Rightarrow \mathbf{P} \mapsto_{\mathbf{I}} \mathbf{Q}$,

(2) $\mathbf{P} \mapsto_{\mathbf{I}} \mathbf{Q} \wedge \mathbf{Q} \mapsto_{\mathbf{I}} \mathbf{R} \Rightarrow \mathbf{P} \mapsto_{\mathbf{I}} \mathbf{R}$,

(3) $W \neq \emptyset \wedge (\forall w \in W\colon \mathbf{P}_w \mapsto_{\mathbf{I}} \mathbf{Q}) \Rightarrow \bigcup_{w \in W} \mathbf{P}_w \mapsto_{\mathbf{I}} \mathbf{Q}$.

Lemma 2.5 (Semantic correctness). $\mapsto$ *is semantically correct w.r.t.* leads-to, *i.e.*,

$$\mathbf{P} \mapsto_{\mathbf{I}} \mathbf{Q} \Rightarrow \mathbf{P} \text{ leads-to}_{\mathbf{I}} \mathbf{Q}.$$

Proof (*sketch*). The proof is straightforward by induction on $\mapsto$. □

3. Semantic completeness of transfinite induction

This section is devoted to the proof of semantic completeness of a transfinite induction principle for $\mapsto$. A similar principle is used frequently in the applications of UNITY logic, and is proved in [5, pp. 72ff.] to follow from closure under disjunction (see [5, p. 52], here Definition 2.4(2)(3)). Besides the theoretical interest shown in this paper, such a principle of transfinite induction also is of great practical value in proving program properties. In fact, in [5] closure of $\mapsto$ under disjunction appears to be used only via applications of transfinite induction.

Our version of transfinite induction is less ad hoc than the principle formulated by Chandy and Misra. Instead of requiring a function mapping states to elements of

a well-founded set, we assume just some classification of the relevant states by means of a well-founded family of sets of states.

The results of this section hold for an arbitrary transition system $\mathscr{T}$ in the sense of Definition 2.1, which is assumed to be given. The corresponding notational conventions are explained in Section 2.

Proposition 3.1 (Principle of transfinite induction for $\mapsto$). *Let $\prec$ be a well-founded relation on a nonempty set W, and $\mathbf{I}$, $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{R}_w$ $(w \in W)$ be sets of states s.t.*
(1) $\mathbf{I} \cap \mathbf{P} \subseteq \mathbf{Q} \cup \bigcup_{w \in W} \mathbf{R}_w$,
(2) $\forall w \in W$: $\mathbf{R}_w \mapsto_{\mathbf{I}} \mathbf{Q} \cup \bigcup_{u \prec w} \mathbf{R}_u$.
Then the following holds:

$$\mathbf{P} \mapsto_{\mathbf{I}} \mathbf{Q}.$$

Proof (*sketch*). The proof is similar to the proof given in [5, pp. 72ff.]. First we show by transfinite induction on $\prec$ that for every $w \in W$: $\mathbf{R}_w \mapsto_{\mathbf{I}} \mathbf{Q}$. A final application of Definition 2.4 (2) (3) yields $\bigcup_{w \in W} \mathbf{R}_w \mapsto_{\mathbf{I}} \mathbf{Q}$, from which the claim follows. The implication and cancellation theorems of [5, p. 64f.] are used as lemmas. □

Semantic completeness of $\mapsto$ w.r.t. leads-to is a corollary of the following strong converse of this proposition. Observe that only ensures relations are used in the hypotheses of this application of the transfinite induction principle for $\mapsto$.

Theorem 3.2 (Semantic completeness of transfinite induction). *Let $\mathbf{I}$, $\mathbf{P}$ and $\mathbf{Q}$ be sets of states s.t. $\mathbf{P}$ leads-to$_{\mathbf{I}}$ $\mathbf{Q}$, then one can define a well-founded partial order $(W, \prec)$ and a family $(\mathbf{R}_w)_{w \in W}$ of sets of states s.t. the following hold:*
(1) $\mathbf{I} \cap \mathbf{P} \subseteq \mathbf{Q} \cup \bigcup_{w \in W} \mathbf{R}_w$,
(2) $\forall w \in W$: $\mathbf{R}_w$ ensures$_{\mathbf{I}}$ $\mathbf{Q} \cup \bigcup_{u \prec w} \mathbf{R}_u$.

The proof of this theorem uses the execution tree of a transition system. To define this notion formally, we first introduce some terminology and notation.

Terminology and notation. For a nonempty set M, M^* denotes the set of finite sequences (words) over M. $w, w', \ldots$ range over M^*. $(\,)$ denotes the empty sequence. The following standard functions and relations on sequences are used:

$$(x_0, \ldots, x_{m-1}) @ (y_0, \ldots, y_{n-1}) \stackrel{\text{def}}{=} (x_0, \ldots, x_{m-1}, y_0, \ldots, y_{n-1}),$$

$$|(x_0, \ldots, x_{n-1})| \stackrel{\text{def}}{=} n,$$

$$\mathrm{last}((x_0, \ldots, x_{n-1})) \stackrel{\text{def}}{=} x_{n-1},$$

$$((x_0, \ldots, x_{n-1}))_i \stackrel{\text{def}}{=} x_i \quad \text{if } i < n,$$

$$(x_0, \ldots, x_{n-1}) \lceil_i \stackrel{\text{def}}{=} \begin{cases} (x_0, \ldots, x_{i-1}) & \text{if } i \leqslant n, \\ (x_0, \ldots, x_{n-1}) & \text{if } i > n, \end{cases}$$

$$w \geqslant^* w' \stackrel{\text{def}}{\Leftrightarrow} |w| \leqslant |w'| \wedge w' \lceil_{|w|} = w.$$

$\geqslant^*$ is a partial order and is called the *initial segment ordering* on M^*. $>^*$ denotes the corresponding strict partial order, and $\leqslant^*$, $<^*$ are, of course, the converses. For any $X \subseteq M^*$ the respective restrictions to X are denoted by $\geqslant^*_X$, $>^*_X$, $\leqslant^*_X$, $<^*_X$.

$\mathbb{N}$ denotes the set of natural numbers. Throughout the paper 'number' is taken to mean 'natural number'.

Following the practice of recursion theory, we think of trees as being represented by sets of finite sequences over an underlying set M, and infinite paths in trees as represented by infinite sequences over M. An *infinite sequence over M* is a sequence of elements of M indexed by $\mathbb{N}$. We let $\pi, \pi', \pi_1, \ldots$ range over such infinite sequences. The following notation is used for initial segments of $\pi = (\pi_n)_{n \in \mathbb{N}}$:

$$\pi\lceil_n = (\pi_0, \ldots, \pi_{n-1}).$$

Definition 3.3 (*Tree*). Let M be a nonempty set.
(1) Tr is a *tree* over $M \overset{\text{def}}{\Leftrightarrow} Tr \subseteq M^* \wedge \forall w \in Tr\colon \forall w' \in M^*\ (w' \geqslant^* w \Rightarrow w' \in Tr)$.
(2) π is an *infinite path* in $Tr \overset{\text{def}}{\Leftrightarrow} \forall n \in \mathbb{N}\colon \pi\lceil_n \in Tr$.
(3) A tree Tr is called *well-founded* iff there is no infinite path in Tr.
(Observe that this is equivalent to well-foundedness of $<^*_{Tr}$.)

A tree in this sense has no explicit representation of edges. Rather these are given implicitly by the tacit understanding that an edge goes from $w \in Tr$ to every $w' \in Tr$ with $w' <^*_{Tr} w$ and $|w'| = |w| + 1$.

We are now ready to define the notion of an execution tree. Since trees are represented without explicit edges, one cannot label edges by transitions. Instead we use the nodes of the tree to denote states and transitions alternatingly. Apart from the full execution tree, restrictions are considered. These are of the form that the initial states belong to a given set, and the whole tree avoids some other set. Execution sequences receive a formal representation as paths in the execution tree, called execution paths.

Definition 3.4 (*Execution tree*). (1) $Exec_{\mathscr{T}}$ – the (full) *execution tree* of $\mathscr{T}$ – is defined by $w \in Exec_{\mathscr{T}} \overset{\text{def}}{\Leftrightarrow} \forall i < |w|\colon (i \text{ even} \Rightarrow (w)_i \in \mathbf{S}) \wedge (i \text{ odd} \Rightarrow (w)_i \in \mathscr{T} \wedge (w)_{i+1} = (w)_i((w)_{i-1}))$.

(2) For $\mathbf{P}, \mathbf{Q} \subseteq \mathbf{S}$ a *restricted execution tree* of $\mathscr{T}$ is defined by $w \in Exec_{\mathscr{T}}\lceil \mathbf{P}|\mathbf{Q} \overset{\text{def}}{\Leftrightarrow} w \in Exec_{\mathscr{T}} \wedge (w \neq (\,) \Rightarrow (w)_0 \in \mathbf{P} \wedge \forall i < |w|\colon (i \text{ even} \Rightarrow (w)_i \notin \mathbf{Q}))$.

(3) An infinite path in the execution tree of $\mathscr{T}$ is called an *execution path* of $\mathscr{T}$.

(4) An execution path π of $\mathscr{T}$ is called *fair* iff for every transition $\tau \in \mathscr{T}$ there are infinitely many $i \in \mathbb{N}$ s.t. $\pi(2i+1) = \tau$.

In order to prove completeness of transfinite induction as formulated in Theorem 3.2, one has to define suitable sets $\mathbf{R}_w$, which encode information as to which transition is a 'helpful direction' in the sense of the ensures notion (see Theorem 3.2(2)). This is achieved by introducing an explicit book-keeping of the transitions which have been used at certain points in the execution tree.

Definition 3.5. The function $\mathrm{dir}: Exec_{\mathscr{T}} \to \wp(\mathscr{T})$ denotes the power set of $\mathscr{T}$) is defined by recursion on the length of sequences as follows:

$$\mathrm{dir}(( )) \stackrel{\mathrm{def}}{=} \emptyset,$$

$$\mathrm{dir}((w_0, \ldots, w_{2n}, \tau)) \stackrel{\mathrm{def}}{=} \mathrm{dir}((w_0, \ldots, w_{2n})) \cup \{\tau\},$$

$$\mathrm{dir}((w_0, \ldots, w_{2n})) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{dir}((w_0, \ldots, w_{2n-1})) & \text{if } \mathrm{dir}((w_0, \ldots, w_{2n-1})) \neq \mathscr{T}, \\ \emptyset & \text{if } \mathrm{dir}((w_0, \ldots, w_{2n-1})) = \mathscr{T}. \end{cases}$$

dir is a book-keeping device. dir accumulates the transitions used along a path in the tree, being reset to the empty set, when all transitions have been used. Based on this function one defines an ordering $\ll$ on the execution tree, which captures the idea that progress in an execution has been made, when dir reports a change.

Definition 3.6. (1) $w \in Exec_{\mathscr{T}}$ is *critical* $\stackrel{\mathrm{def}}{\Leftrightarrow} |w|$ *even* $\wedge\ |w| \geqslant 2 \wedge \mathrm{dir}(w\lceil_{|w|-1}) \neq \mathrm{dir}(w)$.

(2) The strict partial order $\ll$ on $Exec_{\mathscr{T}}$ is defined by

$$w' \ll w \stackrel{\mathrm{def}}{\Leftrightarrow} w' <^* w \ \wedge\ \exists i (|w| < i \leqslant |w'| \wedge w'\lceil_i \text{ is critical}).$$

An immediate consequence of these definitions is the following lemma.

Lemma 3.7. (1) *An execution path of $\mathscr{T}$ is fair if and only if it is descending w.r.t. $\ll$.*

(2) *An execution path of $\mathscr{T}$ is a fair execution path from* **P** *avoiding* **Q** *if and only if it is an infinite path in $Exec_{\mathscr{T}}\lceil \mathbf{P} | \mathbf{Q}$ descending w.r.t. $\ll$.*

Proof of Theorem 3.2. Assume to be given sets of states **I**, **P** and **Q** s.t. **P** leads-to$_{\mathbf{I}}$ **Q** holds. By virtue of this assumption together with Lemma 3.7, the following defines a well-founded partial order $(W, \prec)$:

$$W \stackrel{\mathrm{def}}{=} Exec_{\mathscr{T}}\lceil (\mathbf{I} \cap \mathbf{P}) | \mathbf{Q} \cap \{w \mid |w| \text{ odd}\},$$

$$\prec \stackrel{\mathrm{def}}{=} \ll_W.$$

We now associate with every $w \in W$ a set $\mathbf{R}_w$ of states of 'equal rank' enabling us to find a 'helpful direction' for states in $\mathbf{R}_w$ as follows:

$$\mathbf{R}_w \stackrel{\mathrm{def}}{=} \{\mathrm{last}(w') \mid w' \in W \ \wedge\ w' \leqslant^* w \wedge \neg w' \ll w\}.$$

To see that $\mathbf{I} \cap \mathbf{P} \subseteq \mathbf{Q} \cup \bigcup_{w \in W} \mathbf{R}_w$ (i.e. Theorem 3.2(1)) holds, it suffices to observe that for a state $\sigma \in (\mathbf{I} \cap \mathbf{P}) \setminus \mathbf{Q}$ the one-element sequence (σ) satisfies $(\sigma) \in W$, and hence $\sigma \in \mathbf{R}_{(\sigma)}$.

To prove $\forall w \in W$: $\mathbf{R}_w$ ensures$_{\mathbf{I}}$ $\mathbf{Q} \cup \bigcup_{u \prec w} \mathbf{R}_u$ (i.e. Theorem 3.2(2)), let $w \in W$ be given. It suffices to prove the following claim.

Claim.

$$\forall\tau\in\mathscr{T}\colon \forall\sigma\in(\mathbf{I}\cap\mathbf{R}_w)\setminus\mathbf{Q}\colon \tau(\sigma)\in\mathbf{Q}\cup\bigcup_{u\leqslant w}\mathbf{R}_u$$

and

$$\exists\tau\in\mathscr{T}\colon \forall\sigma\in(\mathbf{I}\cap\mathbf{R}_w)\setminus\mathbf{Q}\colon \tau(\sigma)\in\mathbf{Q}\cup\bigcup_{u\prec w}\mathbf{R}_u.$$

Consider an arbitrary transition $\tau\in\mathscr{T}$, and an arbitrary $\sigma\in(\mathbf{I}\cap\mathbf{R}_w)\setminus\mathbf{Q}$. If $\tau(\sigma)\in\mathbf{Q}$, we are done. If not, we look at some $w'\in W$ s.t. $w'\leqslant^* w$, $\neg w'\ll w$, and $\mathsf{last}(w')=\sigma$. Note that $\mathsf{dir}(w')=\mathsf{dir}(w)$ holds by definition of $\ll$, and $w'@(\tau,\tau(\sigma))\in W$ holds by virtue of $\tau(\sigma)\notin\mathbf{Q}$. We distinguish two cases.

Case 1: $\tau\in\mathsf{dir}(w)$. $\mathsf{dir}(w'@(\tau,\tau(\sigma)))=\mathsf{dir}(w')=\mathsf{dir}(w)$, so $w'@(\tau,\tau(\sigma))\leqslant^* w$ and $\neg w'@(\tau,\tau(\sigma))\ll w$, and hence $\tau(\sigma)\in\mathbf{R}_w$.

Case 2. $\tau\in\mathscr{T}\setminus\mathsf{dir}(w)$. $w'@(\tau)$ is critical, so $w'@(\tau,\tau(\sigma))\prec w$ and $\tau(\sigma)\in\mathbf{R}_{w'@(\tau,\tau(\sigma))}$.

To complete the proof of the claim, note that for at least one transition τ, case 2 occurs. This is because $|w|$ is odd, and so $\mathsf{dir}(w)\neq\mathscr{T}$ (cf. Definition 3.5). Furthermore, observe that the same τ works for all $\sigma\in(\mathbf{I}\cap\mathbf{R}_w)\setminus\mathbf{Q}$. □

Corollary 3.8 (Semantic correctness and completeness). *Let* **I**, **P**, *and* **Q** *be sets of states. Then the following are equivalent*:

(1) $\mathbf{P}\mapsto_{\mathbf{I}}\mathbf{Q}$.

(2) $\mathbf{P}\,\mathsf{leads\text{-}to}_{\mathbf{I}}\,\mathbf{Q}$.

(3) *There are a well-founded partial order* $(W,\prec)$ *and a family* $(\mathbf{R}_w)_{w\in W}$ *of sets of states s.t.*

(a) $\mathbf{I}\cap\mathbf{P}\subseteq\mathbf{Q}\cup\bigcup_{w\in W}\mathbf{R}_w$,

(b) $\forall w\in W\colon \mathbf{R}_w\,\mathsf{ensures}_{\mathbf{I}}\,\mathbf{Q}\cup\bigcup_{u\prec w}\mathbf{R}_u$.

(4) *There are a nonempty set* W, *a well-founded relation* $\prec$ *on* W, *and a family* $(\mathbf{R}_w)_{w\in W}$ *of sets of states s.t.*

(a) $\mathbf{I}\cap\mathbf{P}\subseteq\mathbf{Q}\cup\bigcup_{w\in W}\mathbf{R}_w$,

(b) $\forall w\in W\colon \mathbf{R}_w\mapsto_{\mathbf{I}}\mathbf{Q}\cup\bigcup_{u\prec w}\mathbf{R}_u$.

Proof. (1) ⇒ (2) holds by Lemma 2.5. (2) ⇒ (3) holds by Theorem 3.2. (3) ⇒ (4) holds by Definition 2.4 (2) (1). (4) ⇒ (1) holds by Proposition 3.1. □

4. Recursion–theoretic complexity

In this section the recursion-theoretic complexity of leads-to and of the constructions of Section 3 is analyzed. It turns out that the *fairness condition* gives rise to Π^1_1-completeness of the halting problem for fair executions of recursive transition systems as well as to the necessity of using arbitrarily large recursive ordinals as ordinal heights of well-founded partial orders in the transfinite induction principle for $\mapsto$.

This is no surprise, since analogous results are well known for programming concepts involving assignment of a random natural number (e.g, [4, 3]), and also it is known that *countable nondeterminism* and *fairness* conditions for finite nondeterminism – in the context of communicating processes also called 'finite delay property' – are closely related (e.g., [2, 3, Section 1]).

Computations using random assignment have an infinitely branching execution tree, and one sees immediately, how large recursive ordinals enter the picture. So it may be instructive to repeat in the context of UNITY a simple construction showing how to mimick random assignment (e.g., [4, p. 128, 3, p. 724]). This construction is used repeatedly in later, more complicated programs.

In our programs we denote by nat the datatype of natural numbers.

Program $\mathscr{RA}$
declare x, c : nat
initially $x=0 \wedge c=0$
assign
(a) $x := x+1$ if $c=0$
(b) $c := 1$
end $\mathscr{RA}$

Clearly every fair execution sequence of program $\mathscr{RA}$ reaches a fixed point, and when this fixed point is reached a 'random number has been assigned to x' in the following sense: for every number $\hat{x}$, there is a fair execution sequence of $\mathscr{RA}$ s.t. at the fixed point reached by $\mathscr{RA}$ the value of x is $\hat{x}$.

Conversely, random assignment can be used to implement fair schedulers, whereby fair execution can be simulated in languages with random assignment. This is studied extensively in [2].

The main ideas for the UNITY programs constructed to prove the results of this section are the same as in [3, Section 5]. Here it is shown that these ideas work also with the restricted means of UNITY lacking control constructs and demanding total functions in the assignments (whereby partial recursive function application $\{\cdot\}(\cdot)$ cannot be used in assignments).

Our results show that despite of these restrictions UNITY has essentially the same expressive and computational power as earlier more complicated nondeterministic languages.

4.1. Prerequisites from recursion theory

For this section some familiarity with the arithmetical and analytical hierarchy of recursion theory is helpful. The basic notions and results can be found in, for example [20, pp. 361–387]. For convenience, we summarize what is needed here.

The *arithmetical* and *analytical hierarchy* classify relations over the natural numbers. More generally, also sets of numbers and/or (unary) numerical functions are

admitted as parameters in these relations. Such a relation is called *arithmetical* if it is definable from a recursive relation by quantification over numbers. It is called *analytical* if it is definable from an arithmetical relation by quantification over numerical functions and/or sets of numbers. An arithmetical relation is in the complexity class Σ_n^0/Π_n^0 $(n \geqslant 1)$ if the quantifier prefix of an arithmetical definition begins with an existential/universal number quantifier and has at most $n-1$ quantifier alternations in it. An analytical relation is in the complexity class Σ_n^1/Π_n^1 $(n \geqslant 1)$ if the function and set quantifier prefix of an analytical definition begins with an existential/universal function or set quantifier and has at most $n-1$ alternations of the kind of function and set quantifier in it (number quantifiers are ignored in the analytical classification). Σ_1^0 is the class of *semi-recursive* relations (also called *recursively enumerable*, if no function or set parameters are present).

For each of these complexity classes there are normal forms. One such normal form is the so-called many-one-reduction (m-reduction) to a fixed set in a given complexity class, which is then called complete for this class.

Definition 4.1 (*Many-one-reducibility and completeness*). (1) Let A, B be subsets of $\mathbb{N}$. A is called *m-reducible* to B iff there is a total recursive function h s.t.

$$\forall x \in \mathbb{N} \quad (x \in \mathsf{A} \Leftrightarrow h(x) \in \mathsf{B}).$$

(2) Let $\mathscr{K}$ be a class of subsets of $\mathbb{N}$ and $\mathsf{K} \subseteq \mathbb{N}$. K is called *$\mathscr{K}$-complete* iff $\mathsf{K} \in \mathscr{K}$ and every $\mathsf{A} \in \mathscr{K}$ is m-reducible to K.

Of particular interest for this section is the fact that the set of indices of well-founded recursive trees to be defined below is Π_1^1-complete.

In recursion theory, trees over $\mathbb{N}$ (in the sense of Definition 3.3) are viewed as subsets of $\mathbb{N}$ by virtue of encoding finite sequences of numbers. We assume a standard primitive recursive coding function

$$\langle\ \rangle : \mathbb{N}^* \to \mathbb{N}$$

(e.g., [20, pp. 26f., 88f.]) and translate the definitions about trees via this encoding. The same symbols continue to be used, e.g., $\geqslant^*$ denotes the *initial segment ordering*, which is now viewed as a relation on $\mathbb{N}$ rather than on $\mathbb{N}^*$.[7]

Furthermore a standard enumeration of the partial recursive (unary) functions obtained by formalizing some universal computing device is assumed, and as usual $\{e\}$ denotes the partial recursive function with index e, $\simeq$ denotes partial equality, $\downarrow$ denotes definedness, and $\uparrow$ undefinedness (e.g., [20, pp. 127–133]).

[7] Use of angle brackets instead of ordinary brackets indicates what is meant. Of course, we write $\langle x_0, \ldots, x_{n-1} \rangle$ rather than $\langle (x_0, \ldots, x_{n-1}) \rangle$ for the number encoding the finite sequence $(x_0, \ldots, x_{n-1})$.

Definition 4.2 (*Recursive tree*). (1) Each number e is the index of a tree Tr_e by virtue of the following definition:

$$\forall n \in \mathbb{N} \quad (n \in Tr_e \overset{\text{def}}{\Leftrightarrow} \forall n' \geqslant^* n\colon \{e\}(n) \simeq 0).$$

(2) Tr_e is called a *recursive tree* iff $\{e\}$ *is a total function.*

The following is a standard result from recursion theory (e.g. [20, Corollary IV.2.16, p. 383]).

Definition and Lemma 4.3 (Tree Theorem).

$$\mathsf{WfTree} \overset{\text{def}}{=} \{e \mid Tr_e \textit{ is a well-founded recursive tree}\} \textit{ is } \Pi^1_1\textit{-complete.}$$

4.2. The halting problem

In this section the complexity of the halting problem for fair executions of UNITY programs, or more generally of transition systems, is analyzed. The notion of final state in the halting problem is replaced by the notion of fixed point.

The general halting problem for recursive transition systems is shown to be Π^1_1, and a specific UNITY program is exhibited whose halting problem is Π^1_1-complete.

To move into the realm of recursion theory over the natural numbers, the set of states $\mathbf{S}$ is assumed to be countable and decidable, so modulo some encoding one can assume w.l.o.g. that $\mathbf{S}$ is a recursive subset of $\mathbb{N}$. Moreover the transitions of the transition system are required to be labelled uniquely by numbers. This enables us to replace transitions in the execution tree by their labels. Thereby and by coding of finite sequences, also the execution tree of a transition system is a subset of $\mathbb{N}$.

Definition 4.4 (Recursive transition system). (1) $\mathscr{T} = (\mathbf{S}, \tau_1, \ldots, \tau_n)$ is called a *recursive transition system* over the state space $\mathbf{S}$ iff $\mathbf{S}$ is a recursive, nonempty subset of $\mathbb{N}$, $n \geqslant 1$, and for every $i \in \{1, \ldots, n\}$ the function τ_i is a total recursive function mapping $\mathbf{S}$ to $\mathbf{S}$.

(2) A *UNITY program* $\mathscr{T}$ is called *recursive* iff the transition system induced by $\mathscr{T}$ is recursive.

Remark. Most UNITY programs occurring in practice are recursive. In fact, introducing the notion of a recursive UNITY program almost amounts to splitting hairs. Whether a UNITY program is recursive or not, only depends on the algebra over which the program computes. When the carriers associated with the sorts of the language are recursive, and when the function and predicate symbols of the language denote recursive functions and predicates, then any UNITY program computing over this algebra is automatically recursive. If one wants to use UNITY for specification on abstract levels of system design, however, it seems quite reasonable to admit working with 'abstract UNITY programs' which are not recursive.

Definition 4.5 (*Halting problem*). (1) The set of *fixed points* of a transition system $\mathscr{T}$ is defined as

$$\mathbf{Fix}_{\mathscr{T}} \stackrel{\text{def}}{=} \{\sigma \in \mathbf{S} \mid \forall \tau \in \mathscr{T}: \tau(\sigma) = \sigma\}.$$

(2) The *halting problem* for a transition system $\mathscr{T}$ is defined to be the set

$$\mathbf{Halt}_{\mathscr{T}} \stackrel{\text{def}}{=} \{\sigma \in \mathbf{S} \mid \{\sigma\} \text{ leads-to}_{\mathbf{S}} \mathbf{Fix}_{\mathscr{T}}\}.$$

(3) The *general halting problem* for recursive transition systems is defined to be the set

$$\mathsf{Halt} \stackrel{\text{def}}{=} \{(\mathscr{T}, \sigma) \mid \mathscr{T} \text{ recursive transition system } \wedge\ \sigma \in \mathbf{Halt}_{\mathscr{T}}\}.$$

Remark. $\mathbf{Halt}_{\mathscr{T}}$ is the largest set of states satisfying

$$\mathbf{Halt}_{\mathscr{T}} \text{ leads-to}_{\mathbf{S}} \mathbf{Fix}_{\mathscr{T}}.$$

This follows from $\mathbf{P} \text{ leads-to}_{\mathbf{I}} \mathbf{Q} \Leftrightarrow \forall \sigma \in \mathbf{P}: \{\sigma\} \text{ leads-to}_{\mathbf{I}} \mathbf{Q}$.

For the purpose of analyzing the complexity of the general halting problem for recursive transition systems, we identify such a system $\mathscr{T}$ with a number encoding it, e.g., a code for the sequence consisting of a recursive index for the characteristic function of its state space, and the recursive indices for its transition functions.

Lemma 4.6. *The general halting problem for recursive transition systems is* Π_1^1.

Proof (*sketch*). Being (the code of) a recursive transition system is an arithmetical notion. The description of $\{\sigma\} \text{ leads-to}_{\mathbf{S}} \mathbf{Fix}_{\mathscr{T}}$ begins with a universal function quantifier, namely over fair execution paths. The property of a function of being a fair execution path w.r.t. transition system $\mathscr{T}$ is arithmetical, and also the property of being in $\mathbf{Fix}_{\mathscr{T}}$. □

Theorem 4.7. *There is a recursive UNITY program whose halting problem is* Π_1^1*-complete.*

The program being exhibited to prove this theorem computes over the natural numbers. If it were wanted, we could restrict overselves to the successor function, predecessor function, and characteristic function of equality as primitive functions used in this program.[8] To keep the program small and comprehensible, we prefer, however, to allow some other primitive recursive functions to occur in the assignment

[8] This can be achieved by simulating in UNITY deterministic register machine computations of recursive functions.

statements. Among these are the standard functions on codes of sequences and the following function enum.

By standard techniques of recursion theory, we can assume to have at our disposal a primitive recursive function enum[9] s.t.

$$\mathrm{enum}(e,n,t)=\begin{cases} m+1 & \text{if } \leqslant t \text{ steps of computation of } \{e\}(n) \text{ yield result } m,\\ 0 & \text{if computation of } \{e\}(n) \text{ is not finished within } t \text{ steps.}\end{cases}$$

Definition 4.8. Program $\mathscr{RT}$

declare e,c,n,b,t : nat

initially $c=1 \wedge n=\langle\ \rangle \wedge b=0 \wedge t=0$

assign

(1)	t	:=	$t+1$	if $c>0$
(2)	b	:=	$b+1$	if $c>0$
(3)	c,n,b	:=	$c, n@(b-1), 0$	if $1\leqslant c\leqslant 2 \wedge \mathrm{enum}(e,n,t)=1 \wedge b>0$
		$\sim$	$2, n@(b-1), 0$	if $1\leqslant c\leqslant 2 \wedge \mathrm{enum}(e,n,t)>1 \wedge b>0$
(4)	c	:=	3	if $c=2$
(5)	c	:=	0	if $c=3 \wedge \mathrm{enum}(e,n,t)>0$

end $\mathscr{RT}$

The purpose of the program $\mathscr{RT}$ is to scrutinize with help of the function enum whether the partial recursive function $\{e\}$ defines a well-founded recursive tree Tr_e. Before stating the key lemma about $\mathscr{RT}$ from which Theorem 4.7 follows, let us first describe informally, what happens in a legal execution sequence of this program.

The names of the program variables are intended to be mnemonic: c for 'counter of execution phase', n for 'node of a tree', b for 'branch to be taken next', and t for 'time (=number of computation steps in the sense of enum)'.

The nondeterministic choices of a fair execution sequence lead to the choice of a path in the tree of sequence codes. In the first phase of execution (value of $c=1$), it is successively checked along this path whether for a node n, $\{e\}(n)\simeq 0$. As soon as a node n is found s.t. $\{e\}(n)\downarrow$ and $\{e\}(n)>0$, the value of c is set to 2 to record this fact. In the second phase of execution (value of $c=2$), the path is checked further down for definedness of $\{e\}(n)$. Setting c to 3 by execution of assignment (4), indicates that the path shall not be extended any more for checking. In the third phase of execution (value of $c=3$), the last point of the travelled segment of the path is checked for definedness of $\{e\}(n)$. When this check has succeeded, a fixed point is reached by setting c to 0.

[9] Such a function can be defined easily from the predicate T and the function U occurring in Kleene's normal form theorem, e.g. [20, Theorem II.1.2, p. 129]. Our intuitive formulation 'computation of $\{e\}(n)$ is finished within t steps' is to be made precise by '$T(e,n,t)$'.

Lemma 4.9. $\forall \hat{e} \in \mathbb{N}$: *$Tr_e$ is a well-founded recursive tree* $\Leftrightarrow$ $e=\hat{e} \wedge c=1 \wedge n=\langle\,\rangle \wedge b=0 \wedge t=0$ leads-to$_S$ $c=0$.[10]

Proof. This is proved by the following series of claims.

Claim 1.

$$\sigma \in \mathbf{Fix}_{\mathscr{R}\mathscr{T}} \Leftrightarrow \sigma(c)=0.$$

Proof. When the value of c is positive, assignment (1) increases the value of t. On the other hand, all assignments do nothing, when the value of c is 0. □

Claim 2.

$$c=1 \text{ unless}_S\ c=2 \ \wedge\ c=2 \text{ unless}_S\ c=3 \ \wedge\ c=3 \text{ unless}_S\ c=0$$
$$\wedge\ \text{stable}\ c=0 \ \wedge\ \text{stable}\ 0 \leqslant c \leqslant 3.$$

Proof. The value of c can only change from 1 to 2 by assignment (3), from 2 to 3 by assignment (4), and from 3 to 0 by assignment (4). □

Proof of Lemma 4.9 (*conclusion*). In the following, we fix some number $\hat{e}$ and look at execution segments and execution sequences beginning in the state σ_0 defined by

$$\sigma_0(e)=\hat{e} \ \wedge\ \sigma_0(c)=1 \ \wedge\ \sigma_0(n)=\langle\,\rangle \ \wedge\ \sigma_0(b)=0 \ \wedge\ \sigma_0(t)=0.$$

Claim 3. *If $\langle b_0, \ldots, b_{k-1}\rangle$ is a node s.t. $\forall n' >^* \langle b_0, \ldots, b_{k-1}\rangle$: $\{\hat{e}\}(n')\downarrow$, then there is an execution segment starting in state σ_0 and ending in a state σ with*

$$\sigma(n)=\langle b_0, \ldots, b_{k-1}\rangle \ \wedge\ \sigma(b)=0 \ \wedge\ 1 \leqslant \sigma(c) \leqslant 2.$$

Proof. *By induction on k.* For $k=0$, the one-element segment (σ_0) satisfies the claim. For $k \Rightarrow k+1$, one can continue the segment obtained by induction hypothesis by applying (2) b_k+1 times, then applying (1) at least once and frequently enough s.t. the value $\hat{t}$ of t satisfies $\text{enum}(\hat{e}, \langle b_0, \ldots, b_{k-1}\rangle, \hat{t}) > 0$, and then applying (3) once. □

Claim 4. *If $\{\hat{e}\}$ is not total, then there is a fair execution sequence beginning in state σ_0 s.t. for every state σ in this sequence, $\sigma(c)>0$.*

Proof. Consider $\langle b_0, \ldots, b_{k-1}\rangle$ s.t.

$$\forall n' >^* \langle b_0, \ldots, b_{k-1}\rangle: \{\hat{e}\}(n')\downarrow \ \wedge\ \{\hat{e}\}(\langle b_0, \ldots, b_{k-1}\rangle)\uparrow.$$

[10] By an obvious abuse of notation, we specify a set of states by a formula. Pedantically we should write, for example, $\{\sigma \in \mathbf{S} \mid \sigma(c)=0\}$ instead of simply $c=0$. In Definition 5.1 this abuse will be turned into a use.

Start with an execution segment according to Claim 3 ending in a state, say σ, satisfying $\sigma(n)=\langle b_0,\ldots,b_{k-1}\rangle \wedge 1\leqslant\sigma(c)\leqslant 2$. From σ continue this segment arbitrarily in a fair way. By assumption it holds that $\forall\hat{t}$: $\mathrm{enum}(\hat{e},\langle b_0,\ldots,b_{k-1}\rangle,\hat{t})=0$, and hence n never changes its value and c never gets value 0 after state σ. □

Claim 5. *If $\{\hat{e}\}$ is total, but $Tr_{\hat{e}}$ is not well-founded, then there is a fair execution sequence beginning in state σ_0 s.t. for every state σ in this sequence, $\sigma(c)=1$.*

Proof. Let $\tilde{b}=(b_k)_{k\in\mathbb{N}}$ be an infinite path in $Tr_{\hat{e}}$. Construct an execution sequence by successively applying Claim 3 to the initial segments $\tilde{b}\lceil_k$. By the proof of Claim 3, assignments (1)–(3) occur infinitely often in this execution sequence. By assumption, there are no $k,\hat{t}$ s.t. $\mathrm{enum}(\hat{e},\tilde{b}\lceil_k,\hat{t})>1$, and so in this execution sequence the value of c is never set to 2, and hence never set to 3 or 0. Plugging infinitely many applications of assignments (4) and (5) into this execution sequence at arbitrary places yields a fair execution sequence with the required properties. □

Claim 6. *If there is a fair execution sequence beginning in state σ_0 s.t. for every state σ in this sequence $\sigma(c)>0$, and n changes its value only finitely often, then $\{\hat{e}\}$ is not total.*

Proof. Let such a fair execution sequence be given. There is a state in it s.t. from this state on the value of n never changes. At some later state, say σ, the value of b must be >0 by virtue of an application of assignment (2). From state σ on, the value of b is always >0, since it can be set to 0 only by assignment (3) simultaneously with changing the value of n. By assumption, also the value of c is >0 from state σ on. By executing assignments (3) and (5) after state σ, $\mathrm{enum}(e,n,t)$ is evaluated for ever increasing values of t. This execution never causes a change. This can only be due to the fact that evaluation of $\mathrm{enum}(e,n,t)$ always yields 0. This proves that $\{\hat{e}\}(\sigma(n))$ is undefined, and hence $\{\hat{e}\}$ is not total. □

Claim 7. *If there is a fair execution sequence beginning in state σ_0 s.t. for every state σ in this sequence $\sigma(c)>0$ and n changes its value infinitely often, then $Tr_{\hat{e}}$ is not well-founded.*

Proof. Let such a fair execution sequence π be given. First observe that the value of c can never be 3 (once this were so, the value of n would not be changed thereafter), and so it can never be 2 (once it were 2, it were sometime later set to 3). So the value of c must be 1 for all states of π. The value of n changes due to executions of assignment (3). So the values taken by n in π are the consecutive initial segments $\tilde{b}\lceil_k$ of some infinite path $\tilde{b}$ in the tree of sequence codes. For an arbitrary $k\in\mathbb{N}$, let us look at the state σ in π s.t. $\sigma(n)=\tilde{b}\lceil_k$ and for the next state σ' in π: $\sigma'(n)=\tilde{b}\lceil_{k+1}$. Clearly $\mathrm{enum}(\hat{e},\tilde{b}\lceil_k,\sigma(t))=1$, and so $\{\hat{e}\}(\tilde{b}\lceil_k)\simeq 0$. One concludes: $\forall k$: $\tilde{b}\lceil_k\in Tr_{\hat{e}}$. □

This completes the proof of Lemma 4.9. □

Proof of Theorem 4.7. It suffices to show m-reducibility of WfTree to $\mathbf{Halt}_{\mathscr{RT}}$. Assume the states of the transition system induced by $\mathscr{RT}$ to be represented as code of the 5-tuple of values of the variables e, c, n, b, t. From Lemma 4.9 it follows that

$$\forall \hat{e} \in \mathbb{N} \quad (\hat{e} \in \mathsf{WfTree} \Leftrightarrow \langle \hat{e}, 1, \langle\,\rangle, 0, 0\rangle \in \mathbf{Halt}_{\mathscr{RT}}).$$

This is an m-reduction as required. □

4.3. Computational power

This section is concerned with the computational power of UNITY. Here we restrict attention to UNITY programs computing over the natural numbers. Such programs can be viewed as computing a numerical function in the following way.

Let $\mathscr{T}$ be a UNITY program whose program variables are of sort nat, and let $x_1, \ldots, x_n$ be the ordered sequence of those program variables which are not initialized in the initially-section of $\mathscr{T}$. (W.l.o.g. one can assume that there is at least one such program variable.) The function computed by $\mathscr{T}$ is defined as follows. Given numbers $\hat{x}_1, \ldots, \hat{x}_n$, a legal execution sequence of $\mathscr{T}$ is started with $\hat{x}_1, \ldots, \hat{x}_n$ as initial values of $x_1, \ldots, x_n$. When this execution sequence reaches a fixed point, the value of x_1 at that fixed point is considered to be a value of the function computed by the program $\mathscr{T}$. Different execution sequences can lead to different values of x_1, so the function being computed is a *nondeterministic function*. Put differently, $\mathscr{T}$ computes the relation 'the value of x_1 is an output of a legal execution sequence of $\mathscr{T}$ with the initial values of $x_1, \ldots, x_n$ as input'. This relation is called the *graph* of the function computed by $\mathscr{T}$. As pointed out in Section 1, the function graph alone does not characterize the I/O-behaviour of a program adequately. One further has to look at the *halting set*, i.e., that subset of the function's *domain*, for which *every* fair execution sequence reaches a fixed point. This leads to the following notion of nondeterministic function.

Definition 4.10 (*Nondeterministic function*). A *nondeterministic function* $f: \mathbb{N}^n \to \mathbb{N}$ is given by a set $\mathsf{G}_f \subseteq \mathbb{N}^n \times \mathbb{N}$ and a set $\mathsf{H}_f \subseteq \mathbb{N}^n$ satisfying

$$\mathsf{H}_f \subseteq \mathsf{D}_f \stackrel{\text{def}}{=} \{x \in \mathbb{N}^n \mid \exists y \in \mathbb{N}: (x, y) \in \mathsf{G}_f\}.$$

G_f is called the *graph*, D_f the *domain*, and H_f the *halting set* of f. One writes $y \in f(x)$ for $(x, y) \in \mathsf{G}_f$ and $\uparrow \in f(x)$ for $x \notin \mathsf{H}_f$.

To measure computational power, we determine the complexity of the graph and of the halting set of the nondeterministic numerical functions, which are computable by recursive UNITY programs. We prove that the result of Chandra [4, Theorem 2, p. 129] for computations of nondeterministic register machines also holds for UNITY programs: a nondeterministic numerical function is computable by a recursive UNITY program if and only if its graph is Σ_1^0, and its halting set is a Π_1^1-subset of its domain.

Upper bounds for the complexity of graph and halting set of the function computed by a recursive UNITY program are easily obtained from the analysis of the halting problem in Section 4.2.

Lemma 4.11. *Let $\mathscr{T}$ be a recursive UNITY program computing a nondeterministic numerical function $f: \mathbb{N}^n \to \mathbb{N}$, and let $\mathsf{H}_f \subseteq \mathbb{N}^n$ be the halting set, $\mathsf{G}_f \subseteq \mathbb{N}^n \times \mathbb{N}$ the graph, and $\mathsf{D}_f \subseteq \mathbb{N}^n$ the domain of f. Then*

$$\mathsf{H}_f \text{ is } \Pi_1^1, \quad \mathsf{G}_f \text{ is } \Sigma_1^0, \quad \mathsf{D}_f \text{ is } \Sigma_1^0.$$

Proof. It follows from Lemma 4.6 that H_f is Π_1^1. '$(\hat{x}_1, \ldots, \hat{x}_n, \hat{y}) \in \mathsf{G}_f$' can be described as 'there is a (code of) a segment of a legal execution sequence starting with $\hat{x}_1, \ldots, \hat{x}_n$ as values of $x_1, \ldots, x_n$, and ending in a fixed point with $\hat{y}$ as value of x_1'. This shows that G_f is Σ_1^0. From this, D_f immediately follows to be Σ_1^0 by Definition 4.10. □

Conversely, given an arbitrary Π_1^1-relation $\mathsf{H} \subseteq \mathbb{N}^n$, and a Σ_1^0-relation $\mathsf{G} \subseteq \mathbb{N}^n \times \mathbb{N}$ satisfying

$$\mathsf{H} \subseteq \mathsf{D} \stackrel{\text{def}}{=} \{x \in \mathbb{N}^n \mid \exists y \in \mathbb{N}: (x, y) \in \mathsf{G}\},$$

then there is a recursive UNITY program having H as the halting set, G as the graph, and D as the domain of the nondeterministic function computed by it. To prove this, we make use of the existence of universal Σ_1^0- and Π_1^1-relations and exhibit programs, which are also universal in a certain sense.

An $n+1$-ary Σ_1^0-relation G is taken to be represented by the index g of a partial recursive function s.t.

$$\forall x_1, \ldots, x_n, y \in \mathbb{N} \quad (\mathsf{G}(x_1, \ldots, x_n, y) \Leftrightarrow \{g\}(\langle x_1, \ldots, x_n, y \rangle)\downarrow).$$

The $n+2$-ary relation $\{(g, x_1, \ldots, x_n, y) \in \mathbb{N}^{n+1} \times \mathbb{N} \mid \{g\}(\langle x_1, \ldots, x_n, y \rangle)\downarrow\}$ is universally Σ_1^0 in the sense that it is itself Σ_1^0, and enumerates all $n+1$-ary Σ_1^0-relations (e.g. [20, Theorem II.1.10, p. 134]). Similarly, a refinement of Π_1^1-completeness of WfTree (cf. Section 4.1) gives us a nice enumeration of the class of Π_1^1-relations. By using a universal Π_1^1-set (e.g. [20, Theorem IV.2.9, p. 380]) in the m-reduction of Π_1^1-sets to WfTree (e.g. [20, Corollary, IV.2.16, p. 383]), one can in fact define a primitive recursive function j s.t. for every $n \geqslant 1$ and every n-ary Π_1^1-relation H there is a number h, called an *index of* H s.t.

$$\forall x \in \mathbb{N}^n \quad (\mathsf{H}(x) \Leftrightarrow j(\langle h, x \rangle) \in \mathsf{WfTree}).$$

For the converse of Lemma 4.11, we first solve the simpler problem of finding programs, for which the halting set is the same as the domain of the function computed. For simplicity of notation, we write our programs only for the case $n = 1$.

Definition 4.12. Program $\mathscr{RE}$

declare $g, x, c, z : \mathrm{nat}$
initially $c=4 \wedge z=0$
assign

(7)	z	$:=$	$z+1$	if $c=4$
(8)	c, x, z	$:=$	$0, (z)_0, z$	if $c=4 \wedge \mathrm{enum}(g, \langle x, (z)_0 \rangle, (z)_1) > 0$
		$\sim$	$5, x, 0$	if $c=4 \wedge \mathrm{enum}(g, \langle x, (z)_0 \rangle, (z)_1) = 0$
(9)	c, x, z	$:=$	$c, x, z+1$	if $c=5 \wedge \mathrm{enum}(g, \langle x, (z)_0 \rangle, (z)_1) = 0$
		$\sim$	$0, (z)_0, z$	if $c=5 \wedge \mathrm{enum}(g, \langle x, (z)_0 \rangle, (z)_1) > 0$

end $\mathscr{RE}$

Informally, this program works as follows. First the program guesses $\langle y, t \rangle$ and checks whether $\mathrm{enum}(g, \langle x, y \rangle, t) > 0$. If this is true, one knows that $\{g\}(\langle x, y \rangle)\downarrow$ holds. The program outputs y and enters a fixed point. If $\mathrm{enum}(g, \langle x, y \rangle, t) = 0$, the program systematically searches for $\langle y, t \rangle$ s.t. $\mathrm{enum}(g, \langle x, y \rangle, t) > 0$. If there is a number y s.t. $\{g\}(\langle x, y \rangle)\downarrow$ holds, then such a $\langle y, t \rangle$ will eventually be found, and the program outputs y and enters a fixed point. Otherwise, the program can never reach a fixed point. The main properties of the program are summarized in the following lemma.

Lemma 4.13. *Let $\hat{g}, \hat{x} \in \mathbb{N}$ be given, and let σ_0 be the state with*

$$\sigma_0(g, x, c, z) = (\hat{g}, \hat{x}, 4, 0).$$

Then the following hold for program $\mathscr{RE}$.

(1) *If an execution sequence beginning in σ_0 reaches a fixed point σ, then $\{\hat{g}\}(\langle \hat{x}, \sigma(x) \rangle)\downarrow$.*

(2) *If $\hat{y} \in \mathbb{N}$ is s.t. $\{\hat{g}\}(\langle \hat{x}, \hat{y} \rangle)\downarrow$, then there is a fair execution sequence beginning in σ_0, which reaches a fixed point σ with $\sigma(x) = \hat{y}$.*

(3) σ_0 leads-to$_S$ $\mathbf{Fix}_{\mathscr{RE}} \Leftrightarrow \exists \hat{y} \in \mathbb{N}: \{\hat{g}\}(\langle \hat{x}, \hat{y} \rangle)\downarrow$.

Program $\mathscr{RE}$ computes the nondeterministic function having as graph the universal Σ_1^0-relation $\{(g, x, y) \in \mathbb{N}^2 \times \mathbb{N} \mid \{g\}(\langle x, y \rangle)\downarrow\}$. Its halting set equals its domain. To obtain an arbitrary Π_1^1-subset of the domain as halting set, we combine this program with the program $\mathscr{RT}$ of Section 4.2 in the following way.

Definition 4.14. Program $\mathscr{RU}$

declare $g, h, x, c, n, b, t, z : \mathrm{nat}$
initially $c=6 \wedge n = \langle\ \rangle \wedge b=0 \wedge t=0 \wedge z=0$
assign

(0)	c	$:=$	1	if $c=6$
(1)	t	$:=$	$t+1$	if $c>0$
(2)	b	$:=$	$b+1$	if $c>0$

$$
\begin{array}{lll}
(3)\ c,n,b := & c,n@(b-1),0 & \text{if } 1\leqslant c\leqslant 2\wedge \mathrm{enum}(\jmath(\langle h,x\rangle),n,t)=1\wedge b>0\\
\sim & 2,n@(b-1),0 & \text{if } 1\leqslant c\leqslant 2\wedge \mathrm{enum}(\jmath(\langle h,x\rangle),n,t)>1\wedge b>0\\
(4)\ c := & 3 & \text{if } c=2\\
(5)\ c := & 4 & \text{if } c=3\wedge \mathrm{enum}(\jmath(\langle h,x\rangle),n,t)>0\\
(6)\ c := & 4 & \text{if } c=6\\
(7)\ z := & z+1 & \text{if } c=4\\
(8)\ c,x,z := & 0,(z)_0,z & \text{if } c=4\wedge \mathrm{enum}(g,\langle x,(z)_0\rangle,(z)_1)>0\\
\sim & 5,x,0 & \text{if } c=4\wedge \mathrm{enum}(g,\langle x,(z)_0\rangle,(z)_1)=0\\
(9)\ c,x,z := & c,x,z+1 & \text{if } c=5\wedge \mathrm{enum}(g,\langle x,(z)_0\rangle,(z)_1)=0\\
\sim & 0,(z)_0,z & \text{if } c=5\wedge \mathrm{enum}(g,\langle x,(z)_0\rangle,(z)_1)>0
\end{array}
$$

end $\mathscr{RU}$

Informally, this program works as follows. Depending on which of assignments (0) or (6) is executed first, program $\mathscr{RU}$ executes program $\mathscr{RT}$ with $\jmath(\langle\hat{h},\hat{x}\rangle)$ as value of e, or it executes program $\mathscr{RE}$. In case $\mathscr{RT}$ is mimicked with choices reaching a fixed point, $\mathscr{RU}$ appends an execution of $\mathscr{RE}$. The main properties of the program are summarized in the following lemma.

Lemma 4.15. *Let $\hat{g},\hat{h},\hat{x}\in\mathbb{N}$ be given, and let σ_0 be the state with*

$$\sigma_0(g,h,x,c,n,b,t,z)=(\hat{g},\hat{h},\hat{x},6,\langle\,\rangle,0,0,0).$$

Then the following hold for program $\mathscr{RU}$.

(1) *If an execution sequence beginning in σ_0 reaches a fixed point σ, then $\{\hat{g}\}(\langle\hat{x},\sigma(x)\rangle)\downarrow$.*

(2) *If $\hat{y}\in\mathbb{N}$ is s.t. $\{\hat{g}\}(\langle\hat{x},\hat{y}\rangle)\downarrow$, then there is a fair execution sequence beginning in σ_0, which reaches a fixed point σ with $\sigma(x)=\hat{y}$.*

(3) $\{\sigma_0\}$ leads-to$_S$ $\mathbf{Fix}_{\mathscr{RU}} \Leftrightarrow \jmath(\langle\hat{h},\hat{x}\rangle)\in\mathsf{WfTree}\wedge\exists\hat{y}\colon\{\hat{g}\}(\langle\hat{x},\hat{y}\rangle)\downarrow$.

Proof. (1) and (2) are left to the reader.

$\Rightarrow$ *of* (3): *Case* 1. $\jmath(\langle\hat{h},\hat{x}\rangle)\notin\mathsf{WfTree}$. By Lemma 4.9, one can find a fair execution sequence of $\mathscr{RT}$, which begins in a corresponding initial state, and does not reach a fixed point. From this, a fair execution sequence of $\mathscr{RU}$ is constructed by prefixing an execution of assignment (0), and interleaving infinitely many executions of assignments (6)–(9) at arbitrary places. The resulting execution sequence never comes to a state, where c has one of the values 4, 5, or 0, so it does not reach a fixed point either.

Case 2: $\forall\hat{y}\colon\{\hat{g}\}(\langle\hat{x},\hat{y}\rangle)\uparrow$. By (1), no fair execution sequence of $\mathscr{RU}$ beginning in σ_0 can possibly reach a fixed point.

$\Leftarrow$ *of* (3): Let a fair execution sequence π of $\mathscr{RU}$ beginning in σ_0 be given.

Case 1: *In π assignment* (0) *is executed before* (6). An initial segment of π 'behaves like' an execution sequence of $\mathscr{RT}$. More precisely, one obtains an execution sequence of $\mathscr{RT}$ by cancelling the (dummy) executions of assignments (0) and (6)–(9). So because of $\jmath(\langle\hat{h},\hat{x}\rangle)\in\mathsf{WfTree}$, by Lemma 4.9 there must be an execution of assignment (5), whereby the value of c is set to 4 (in π). From that point on, π 'behaves like' an

execution sequence of $\mathscr{RE}$ (executions of assignments (0)–(6) being dummy). So because of $\exists \hat{y}$: $\{\hat{g}\}(\langle \hat{x}, \hat{y}\rangle)\downarrow$, π reaches a fixed point by virtue of Lemma 4.13(3).

Case 2: *In* π *assignment* (6) *is executed before* (0). The whole execution sequence 'behaves like' an execution sequence of $\mathscr{RE}$. One argues as in the second part of case 1. □

Program $\mathscr{RU}$ computes a nondeterministic function having as graph a universal Σ_1^0-relation, and as halting set a universal Π_1^1-relation. By generalizing $\mathscr{RU}$ to several variables $\boldsymbol{x}$ in place of x, one obtains the following results.

Theorem 4.16 (Universal UNITY program). *For every* $n \geqslant 1$, *there is a recursive UNITY program* $\mathscr{RU}^n$ *with uninitialized program variables* $g, h, x_1, \ldots, x_n$ *computing the universal* $n+2$*-ary nondeterministic function* u *with the following graph* G_u *and halting set* H_u:

$$\mathsf{G}_u = \{(g, h, \boldsymbol{x}, y) \in \mathbb{N}^{n+2} \times \mathbb{N} \mid \{g\}(\langle \boldsymbol{x}, y\rangle)\downarrow\},$$

$$\mathsf{H}_u = \{(g, h, \boldsymbol{x}) \in \mathbb{N}^{n+2} \mid \jmath(\langle h, \boldsymbol{x}\rangle) \in \mathsf{WfTree} \wedge \exists y \in \mathbb{N}\colon \{g\}(\langle \boldsymbol{x}, y\rangle)\downarrow\}.$$

Corollary 4.17 (Computational power of UNITY). *Let* $f\colon \mathbb{N}^n \to \mathbb{N}$ *be a nondeterministic function with graph* $\mathsf{G}_f \subseteq \mathbb{N}^n \times \mathbb{N}$ *and halting set* $\mathsf{H}_f \subseteq \mathbb{N}^n$. *Then the following hold.*

(1) f *can be computed by a recursive UNITY program* $\Leftrightarrow$ H_f *is* Π_1^1 *and* G_f *is* Σ_1^0.

(2) *If* f *is a deterministic partial function, then* f *can be computed by a recursive UNITY program* $\Leftrightarrow$ f *is a partial recursive function.*

Proof. (1) One direction of the equivalence is settled by Lemma 4.11. For the other direction, let an arbitrary Π_1^1-relation $\mathsf{H} \subseteq \mathbb{N}^n$ and a Σ_1^0-relation $\mathsf{G} \subseteq \mathbb{N}^n \times \mathbb{N}$ be given satisfying $\mathsf{H} \subseteq \mathsf{D} = \{\boldsymbol{x} \in \mathbb{N}^n \mid \exists y \in \mathbb{N}\colon (\boldsymbol{x}, y) \in \mathsf{G}\}$. Let $\hat{g}, \hat{h}$ be indices of G, H, respectively, in the sense explained earlier. Let $\mathscr{RU}^n[\hat{g}, \hat{h}/g, h]$ be the program obtained from $\mathscr{RU}^n$ by deleting g, h as program variables and substituting the numerical constants $\hat{g}, \hat{h}$ for them in the program. Then by Theorem 4.16 the function f computed by $\mathscr{RU}^n[\hat{g}, \hat{h}/g, h]$ has the following graph G_f and halting set H_f:

$$\mathsf{G}_f = \{(\boldsymbol{x}, y) \in \mathbb{N}^n \times \mathbb{N} \mid \{\hat{g}\}(\langle \boldsymbol{x}, y\rangle)\downarrow\} = \mathsf{G},$$

$$\mathsf{H}_f = \{\boldsymbol{x} \in \mathbb{N}^n \mid \jmath(\langle \hat{h}, \boldsymbol{x}\rangle) \in \mathsf{WfTree} \wedge \exists y \in \mathbb{N}\colon \{\hat{g}\}(\langle \boldsymbol{x}, y\rangle)\downarrow\} = \mathsf{H} \cap \mathsf{D} = \mathsf{H}.$$

(2) We take f being 'deterministic' to mean that neither $y_1, y_2 \in \mathbb{N}$ with $y_1 \neq y_2$ and $y_1, y_2 \in f(x)$ can exist, nor can $y \in \mathbb{N}$ with $y \in f(x)$ and $\uparrow \in f(x)$. With this understanding, the halting set of f must equal its domain, and so the result follows from (1) by elementary recursion theory (e.g. [20, Proposition II.1.11, p. 135]). □

4.4. Ordinal analysis

The purpose of this section is to analyze the complexity of the constructions in the proof of the Semantic Completeness Theorem 3.2. In particular it is determined, which ordinal heights of well-founded partial orders are needed to guarantee completeness of transfinite induction in the sense of Theorem 3.2.

We make the same assumptions about transition systems as in Section 4.2. Recall the notions of ordinal height and recursive ordinal (e.g., [20, pp. 384–386]).

Definition 4.18 (*Ordinal height and recursive ordinal*). Let W be a set, $\prec$ a well-founded relation on W, and let ON denote the class of all ordinal numbers.

(1) The ordinal height function $\| \|_{\prec}: W \to \mathsf{ON}$ is defined by recursion on $\prec$ as follows.

$$\|w\|_{\prec} \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if w is minimal w.r.t. } \prec, \\ \sup_{u \prec w} (\|w\|_{\prec} + 1) & \text{otherwise.} \end{cases}$$

(2) The *ordinal height* of $\prec$ is defined by

$$\|\prec\| \stackrel{\text{def}}{=} \sup_{w \in W} (\|w\|_{\prec}).$$

(3) α is called a *recursive ordinal* iff α is the ordinal height of a well-founded recursive tree.

Theorem 4.19 (Complexity bounds for transfinite induction). (1) *Let $\mathcal{T}$ be a recursive transition system, and let* $\mathbf{I}$, $\mathbf{P}$ *and* $\mathbf{Q}$ *be recursive sets of states satisfying*

$$\mathbf{P} \text{ leads-to}_{\mathbf{I}} \mathbf{Q},$$

then one can define a well-founded recursive partial order $(W, \prec)$ and a recursively enumerable family $(\mathbf{R}_w)_{w \in W}$ of sets of states s.t. the following hold:

(a) $\mathbf{I} \cap \mathbf{P} \subseteq \mathbf{Q} \cup \bigcup_{w \in W} \mathbf{R}_w$,

(b) $\forall w \in W$: $\mathbf{R}_w$ ensures$_{\mathbf{I}}$ $\mathbf{Q} \cup \bigcup_{u \prec w} \mathbf{R}_u$,

(c) $\|\prec\|$ *is a recursive ordinal.*

(2) *There is a recursive UNITY program s.t. for every recursive ordinal α there is a state σ_0 satisfying*

$$\{\sigma_0\} \text{ leads-to}_{\mathbf{S}} \mathbf{Fix}_{\mathcal{T}},$$

and for every well-founded relation $\prec$ on a set W and family $(\mathbf{R}_w)_{w \in W}$ of sets of states satisfying (1a) *and* (1b) *above with* $\mathbf{I} = \mathbf{S}$, $\mathbf{P} = \{\sigma_0\}$, *and* $\mathbf{Q} = \mathbf{Fix}_{\mathcal{T}}$:

$$\|\prec\| > \alpha.$$

Proof. (1) is proved by inspection of the proof of Theorem 3.2. By $(W, \prec)$ being recursive we mean that the set W as well as the relation $\prec$ are recursive, and by the family $(\mathbf{R}_w)_{w \in W}$ being recursively enumerable we mean that $\sigma \in \mathbf{R}_w$ as a binary relation of σ and w is recursively enumerable. From the definitions one can easily see that

$Exec_{\mathscr{T}}\lceil \mathbf{P}|\mathbf{Q}$, the function dir, the property of being *critical*, W, and the partial orders $\lll$ and $\prec$ are recursive. Finally, the characterization

$$\sigma\in\mathbf{R}_w \Leftrightarrow \exists w'\in W\colon w'\leqslant^* w \wedge \neg w'\lll w \wedge \mathrm{last}(w')=\sigma$$

shows this relation to be recursively enumerable. Claim (1c) follows by standard results of recursion theory from the fact that W and $\prec$ are recursive (e.g., [20, p. 386]).

For the proof of (2), we use the UNITY program $\mathscr{RT}$ of Definition 4.8. Let a recursive ordinal α be given. Then there is a number $\hat{e}$ s.t. $Tr_{\hat{e}}$ is a well-founded recursive tree with ordinal height $>\alpha$. Let σ_0 be the state $\langle \hat{e}, 1, \langle\,\rangle, 0, 0\rangle$ associated with $Tr_{\hat{e}}$ as in the proof of Theorem 4.7. So $\{\sigma_0\}$ leads-to$_{\mathbf{S}}$ $\mathbf{Fix}_{\mathscr{RT}}$ holds. Now assume to be given a set W with well-founded relation $\prec$ on it, and a family $(\mathbf{R}_w)_{w\in W}$ of sets of states satisfying (1a) and (1b) above with $\mathbf{I}=\mathbf{S}$, $\mathbf{P}=\{\sigma_0\}$, and $\mathbf{Q}=\mathbf{Fix}_{\mathscr{RT}}$.

Claim. There is a mapping $\iota: Tr_{\hat{e}}\to W$ s.t. $n'<^* n \Rightarrow \iota(n')\prec\iota(n)$.

Proof. We use the proof of Lemma 4.9 and refer to it in the following. By Claim 3 in that proof, one can define an embedding, say $\jmath$, of the recursive tree $Tr_{\hat{e}}$ into the restricted execution tree $Exec_{\mathscr{RT}}\lceil\{\sigma_0\}|\mathbf{Fix}_{\mathscr{RT}}$ preserving the respective initial segment ordering $<^*$. In the construction of this embedding, one can take care (cf. the proof of Claim 5 in Lemma 4.9) that in each segment of the execution tree appended in the step from a node of the tree $Tr_{\hat{e}}$ to a son node, each of the assignments of program $\mathscr{RT}$ is executed at least once. By induction on the length of n, we now associate with every $n\in Tr_{\hat{e}}$ an element $\iota(n)\in W$ s.t. the final state of $\jmath(n)$ is an element of $\mathbf{R}_{\iota(n)}\setminus(\mathbf{Fix}_{\mathscr{RT}}\cup\bigcup_{u\prec\iota(n)}\mathbf{R}_u)$.

Induction base: $\jmath(\langle\,\rangle)=\langle\sigma_0\rangle$, and the state $\sigma_0=\langle\hat{e},1,\langle\,\rangle,0,0\rangle$ is not a fixed point of $\mathscr{RT}$, so $\sigma_0\in(\mathbf{S}\cap\{\sigma_0\}\setminus\mathbf{Fix}_{\mathscr{RT}})\subseteq\bigcup_{w\in W}\mathbf{R}_w$ (by (1a)). Choose a $\prec$-minimal $w\in W$ s.t. $\sigma_0\in\mathbf{R}_w$, and let $\iota(\langle\,\rangle)\stackrel{\mathrm{def}}{=} w$.

Induction step: Consider the extension from $n\in Tr_{\hat{e}}$ to $n@(b)\in Tr_{\hat{e}}$. Let σ be the final state of $\jmath(n)$, and σ' the final state of $\jmath(n@(b))$. By induction hypothesis $\sigma\in\mathbf{R}_{\iota(n)}\setminus(\mathbf{Fix}_{\mathscr{RT}}\cup\bigcup_{u\prec\iota(n)}\mathbf{R}_u)$, and by (1b) $\mathbf{R}_{\iota(n)}$ ensures$_{\mathbf{S}}$ $\mathbf{Fix}_{\mathscr{RT}}\cup\bigcup_{u\prec\iota(n)}\mathbf{R}_u$. By construction of the embedding $\jmath$, $\sigma'\notin\mathbf{Fix}_{\mathscr{RT}}$, since $n@(b)\in Tr_{\hat{e}}$. Following down the segment of the execution tree leading from $\jmath(n)$ to $\jmath(n@(b))$, one sees by virtue of the given ensures property, that the states on this segment are in $\mathbf{R}_{\iota(n)}$ for some time, and from some point on they are in $\bigcup_{u\prec\iota(n)}\mathbf{R}_u$. So we let $\iota(n@(b))$ be a $\prec$-minimal $u\in W$ s.t. $\sigma'\in\mathbf{R}_u$. □

From this claim it follows that $\|\prec\|\geqslant\|<^*_{Tr_{\hat{e}}}\|>\alpha$. This proves Theorem 4.19. □

5. Towards formalized logic of UNITY

Our analysis of completeness of transfinite induction gives rise to a formalized (finitary) proof system for $\mapsto$ to be proposed in this section.

It is well known from the study of sequential languages that programming logics cannot be complete in the absolute sense, and a fortiori one cannot expect this for a UNITY logic formalized in a finitary proof system. Notions of relative completeness have been introduced to capture completeness properties of program logics like Hoare logic or dynamic logic in an adequate way (e.g. [1, pp. 437–443]). The same will be done here. The notion of relative completeness appropriate for UNITY logic is completeness relative to the set of Π^1_1-formulas valid in an *arithmetical structure*. The notion of an arithmetical structure has been introduced by Moschovakis [19] under the name *acceptable structure* and used by Harel [10] to prove *arithmetical completeness* of his *dynamic logic* (see also [1, p. 442f., 15, p. 821f.]). Essentially a structure is called arithmetical when it contains a first-order definable copy of the standard model of arithmetic and has first-order definable functions allowing coding and decoding of finite sequences.

The set of true Π^1_1-sentences is a set much more complex than the set of true arithmetical sentences showing up in the Hoare logic. However, one cannot get by with less. The results of Section 4 analyzing the recursion-theoretic complexity of UNITY programs reveal that an oracle knowing about some Π^1_1-complete notion has to be used in a relatively complete proof system for $\mapsto$. Our proof system is correct and relatively complete in the mentioned sense. This illustrates that our proof system is satisfactory from a theoretical point of view. In contrast, our system is not suggested to be adequate for practical purposes. In Section 6 we comment on the problem of designing a formalized 'logic of UNITY' which is reasonable also from a practical point of view.

5.1. Syntax and semantics

A UNITY program uses program variables of certain datatypes, in mathematical terms: it computes over a many-sorted algebra. For the purpose of formalizing program properties, we will therefore use an assertion language based on many-sorted, first-order predicate logic. We briefly sketch the logical framework presupposed here.

Conventions and notation. The notion of a *many-sorted, first-order language* $\mathscr{L}_0$ of predicate logic is assumed to be defined as usual. For us it suffices to have finitely many *sorts*, at most countably many *individual constants* of a suitable sort, and at most countably many *function* and *predicate constants* of a suitable *arity* as specified by the *signature* of the language. Also there are for each sort countably many *individual variables* of this sort. Some of the sorts may be distinguished as *sorts with equality*, which means that a binary predicate symbol $=$ over this sort is in $\mathscr{L}_0$. For sorts $s, s_1, \ldots, s_n$, we write $[s_1, \ldots, s_n] \to s$ for the arity of a function symbol with argument sorts $s_1, \ldots, s_n$ and value sort s, and $[s_1, \ldots, s_n]$ for the arity of a predicate symbol with argument sorts $s_1, \ldots, s_n$. For a term t and a sort s of $\mathscr{L}_0$, we write $t : s$ for 't is of sort s'. Similarly, $f : [s_1, \ldots, s_n] \to s$ for 'f is a function symbol of arity $[s_1, \ldots, s_n] \to s$', and $p : [s_1, \ldots, s_n]$ for 'p is a predicate symbol of arity $[s_1, \ldots, s_n]$'. $\equiv$ denotes syntactic identity of expressions. If E is an expression (term or formula), $\boldsymbol{x}$ a vector of variables,

and t a vector of terms of corresponding sorts, then $E[t/x]$ denotes the expression obtained from E by *simultaneous substitution* of the terms t for the variables x (renaming bound variables in E, if necessary, to avoid clashes).

The language $\mathscr{L}_0$ is extended to a *second-order language* $\mathscr{L}_1$ by admitting, for each arity appropriate for the given signature, countably many *predicate variables* of this arity and quantification over predicate variables.[11] Formulas are denoted by capital roman letters from A to R, possibly decorated by indices etc.

A *UNITY program* $\mathscr{T}$ *over* $\mathscr{L}_0$ is a UNITY program whose expressions in the assignments are terms of $\mathscr{L}_0$, and whose boolean expressions in the conditions of assignments are quantifier-free formulas of $\mathscr{L}_0$. The variables of $\mathscr{L}_0$ occurring in the program $\mathscr{T}$ are called its *program variables*. We require that the sorts of the program variables are sorts with equality.

A *many-sorted structure* $\mathscr{A}$ of the signature of $\mathscr{L}_0$ consists of a nonempty carrier (set of individuals) $|\mathscr{A}|_s$ for each sort s, and individuals, functions, and predicates of the appropriate sorts and arities as demanded by the signature. If s is a sort with equality, then the predicate $=:[s,s]$ is required to be interpreted as equality of individuals in the carrier $|\mathscr{A}|_s$.

A *valuation* of $\mathscr{L}_1$ in $\mathscr{A}$ is a function mapping the individual variables to individuals of the appropriate carrier of $\mathscr{A}$, and the predicate variables to subsets of the appropriate cartesian product of carriers of $\mathscr{A}$. The *satisfaction relation* for the language $\mathscr{L}_1$ is assumed to be defined in the usual way. In particular, the quantifiers over predicate variables are interpreted to range over the *full power set* of the appropriate cartesian product of carriers of $\mathscr{A}$. Valuations are denoted by $\varphi, \varphi', \varphi_1, \ldots, \psi, \psi', \psi_1, \ldots$. If φ is a valuation in $\mathscr{A}$, x a vector of variables, and a a vector of $\mathscr{A}$-objects of corresponding sort or arity, then $\varphi[a/x]$ denotes the valuation, which is obtained from φ by changing the values of x to a.

We write

$$\mathscr{A} \models B(\varphi) \quad \text{and} \quad \mathscr{A} \models B$$

for 'B is satisfied in the structure $\mathscr{A}$ under the valuation φ' and 'B is valid in the structure $\mathscr{A}$' (i.e., 'B is satisfied in the structure $\mathscr{A}$ under every valuation'), respectively.

A *UNITY program* $\mathscr{T}$ *over* $\mathscr{A}$ is a UNITY program over the first-order language $\mathscr{L}_0$ of $\mathscr{A}$, which is interpreted in $\mathscr{A}$. This means that the expressions in the assignments of $\mathscr{T}$ are interpreted in $\mathscr{A}$. Thus the *state space* $\mathbf{S}$ associated with this program is the cartesian product $|\mathscr{A}|_{s_1} \times \cdots \times |\mathscr{A}|_{s_k}$, where $d_1 : s_1, \ldots, d_k : s_k$ are the program variables of $\mathscr{T}$. A transition system over $\mathbf{S}$ is induced by this program according to the semantics of assignments as explained in Section 1.

[11] Only predicate variables of arity [nat], where nat is a specific sort of the language, are actually needed later on.

For a state $\sigma \in \mathbf{S}$, we denote by (φ, σ) the valuation obtained from φ by changing the values of the program variables to the values given by the state σ. So

$$\mathscr{A} \models B(\varphi, \sigma)$$

can be read as 'B is satisfied in the structure $\mathscr{A}$ under the valuation φ in state σ'.

For a fixed UNITY program $\mathscr{T}$, $\mathsf{stable}_{\mathscr{T}}$, $\mathsf{unless}_{\mathscr{T}}$, $\mathsf{ensures}_{\mathscr{T}}$, and $\mapsto_{\mathscr{T}}$ are now introduced as logical operators on top of a many-sorted, first-order language of predicate logic. The semantics of these program quantifiers is defined using the corresponding relations on subsets of the state space of the associated transition system.

We assume to be given a many-sorted, first-order language $\mathscr{L}_0$ of predicate logic, and a UNITY program $\mathscr{T}$ over $\mathscr{L}_0$. These are taken to be fixed for the rest of this section. Many-sorted structures $\mathscr{A}$ considered in the sequel are tacitly assumed to be of the signature of $\mathscr{L}_0$.

Remark. By considering a fixed UNITY program we will arrive at the notion of a 'logic of UNITY program $\mathscr{T}$'. One could also consider $\mathscr{T}$ as a parameter in the following development and thus arrive at a 'logic of UNITY' in the sense of a logic of the class of all UNITY programs over a given language. The point of not doing this is not that one *cannot* do this, but rather that one *need not* do this: It is possible to treat a given UNITY program without considering other related UNITY programs like, for example, subprograms. The properties of a UNITY program can be expressed with reference to its constituent conditional assignments directly.

Conventions and notation. Let $d_1 : s_1, \ldots, d_k : s_k$ denote the program variables of $\mathscr{T}$. Assignments of $\mathscr{T}$ are denoted by c, possibly decorated by indices. When $\mathscr{T}$ is interpreted in a many-sorted structure $\mathscr{A}$, then the induced transition system is denoted by $\mathscr{T}(\mathscr{A})$. Suppressing explicit reference to $\mathscr{A}$, its state space is denoted by $\mathbf{S}$, and for an assignment c, the transition function induced by c in $\mathscr{T}(\mathscr{A})$ is denoted by τ_c.

Definition 5.1 (*Language of UNITY program $\mathscr{T}$*). (1) *Syntax.* $\mathscr{L}_0^{\mathscr{T}}$ – the first-order language of the UNITY program $\mathscr{T}$ – is obtained from $\mathscr{L}_0$ by adding four new logical operators $\mathsf{stable}_{\mathscr{T}}$, $\mathsf{unless}_{\mathscr{T}}$, $\mathsf{ensures}_{\mathscr{T}}$, and $\mapsto_{\mathscr{T}}$, and formulas according to the following stipulations:

$$A, B, I \text{ formulas of } \mathscr{L}_0 \Rightarrow \begin{array}{l} \mathsf{stable}_{\mathscr{T}}\, I, \\ A\ \mathsf{unless}_{\mathscr{T},I}\, B, \\ A\ \mathsf{ensures}_{\mathscr{T},I}\, B, \\ A \mapsto_{\mathscr{T},I} B, \end{array} \quad \text{formulas of } \mathscr{L}_0^{\mathscr{T}}.$$

These logical operators are called *program quantifiers*, and the formulas introduced by these clauses are called *program formulas*. The program quantifiers *bind* the program variables.[12]

(2) *Semantics.* Let $\mathscr{A}$ be a many-sorted structure, and let A, B, C, and I be formulas of $\mathscr{L}_0$.

(a) For a valuation φ in $\mathscr{A}$: $[C]_\varphi \stackrel{\text{def}}{=} \{\sigma \in \mathbf{S} \mid \mathscr{A} \models C(\varphi, \sigma)\}$.

(b) *Satisfaction Relation. For a valuation φ in $\mathscr{A}$:*

$$\mathscr{T}(\mathscr{A}) \models \mathsf{stable}_{\mathscr{T}}\, I(\varphi) \stackrel{\text{def}}{\Leftrightarrow} \mathsf{stable}\,[I]_\varphi \quad \textit{in } \mathscr{T}(\mathscr{A}),$$

$$\mathscr{T}(\mathscr{A}) \models A\ \mathsf{unless}_{\mathscr{T},I}\, B(\varphi) \stackrel{\text{def}}{\Leftrightarrow} [A]_\varphi\ \mathsf{unless}_{[I]_\varphi}[B]_\varphi \quad \textit{in } \mathscr{T}(\mathscr{A}),$$

$$\mathscr{T}(\mathscr{A}) \models A\ \mathsf{ensures}_{\mathscr{T},I}\, B(\varphi) \stackrel{\text{def}}{\Leftrightarrow} [A]_\varphi\ \mathsf{ensures}_{[I]_\varphi}[B]_\varphi \quad \textit{in } \mathscr{T}(\mathscr{A}),$$

$$\mathscr{T}(\mathscr{A}) \models A \mapsto_{\mathscr{T},I} B(\varphi) \stackrel{\text{def}}{\Leftrightarrow} [A]_\varphi\ \mathsf{leads\text{-}to}_{[I]_\varphi}[B]_\varphi \quad \textit{in } \mathscr{T}(\mathscr{A}).$$

(c) *Validity.*

$$\mathscr{T}(\mathscr{A}) \models \mathsf{stable}_{\mathscr{T}}\, I \stackrel{\text{def}}{\Leftrightarrow} \forall\varphi\colon \mathscr{T}(\mathscr{A}) \models \mathsf{stable}_{\mathscr{T}}\, I(\varphi),$$

$$\mathscr{T}(\mathscr{A}) \models A\ \mathsf{unless}_{\mathscr{T},I}\, B \stackrel{\text{def}}{\Leftrightarrow} \forall\varphi\colon \mathscr{T}(\mathscr{A}) \models A\ \mathsf{unless}_{\mathscr{T},I}\, B(\varphi),$$

$$\mathscr{T}(\mathscr{A}) \models A\ \mathsf{ensures}_{\mathscr{T},I}\, B \stackrel{\text{def}}{\Leftrightarrow} \forall\varphi\colon \mathscr{T}(\mathscr{A}) \models A\ \mathsf{ensures}_{\mathscr{T},I}\, B(\varphi),$$

$$\mathscr{T}(\mathscr{A}) \models A \mapsto_{\mathscr{T},I} B \stackrel{\text{def}}{\Leftrightarrow} \forall\varphi\colon \mathscr{T}(\mathscr{A}) \models A \mapsto_{\mathscr{T},I} B(\varphi).$$

Satisfaction of $\mathsf{stable}_{\mathscr{T}}$, $\mathsf{unless}_{\mathscr{T}}$, and $\mathsf{ensures}_{\mathscr{T}}$ formulas in $\mathscr{T}(\mathscr{A})$ can be expressed by satisfaction in $\mathscr{A}$ using the definition of the semantic notions in Definitions 2.2 and 2.4.

Lemma 5.2. *Let $\mathscr{A}$ be a many-sorted structure.*

(1) $$\mathscr{T}(\mathscr{A}) \models \mathsf{stable}_{\mathscr{T}}\, I(\varphi) \Leftrightarrow \bigwedge_{c \in \mathscr{T}} \forall\sigma \in \mathbf{S}(\mathscr{A} \models I(\varphi, \sigma) \Rightarrow \mathscr{A} \models I(\varphi, \tau_c(\sigma))),$$

(2) $$\mathscr{T}(\mathscr{A}) \models A\ \mathsf{unless}_{\mathscr{T},I}\, B(\varphi) \Leftrightarrow \mathscr{T}(\mathscr{A}) \models \mathsf{stable}_{\mathscr{T}}\, I(\varphi) \wedge \bigwedge_{c \in \mathscr{T}} \forall\sigma \in \mathbf{S}(\mathscr{A} \models I \wedge A \wedge \neg B(\varphi, \sigma) \Rightarrow \mathscr{A} \models A \vee B(\varphi, \tau_c(\sigma))),$$

(3) $$\mathscr{T}(\mathscr{A}) \models A\ \mathsf{ensures}_{\mathscr{T},I}\, B(\varphi) \Leftrightarrow \mathscr{T}(\mathscr{A}) \models A\ \mathsf{unless}_{\mathscr{T},I}\, B(\varphi) \wedge \bigvee_{c \in \mathscr{T}} \forall\sigma \in \mathbf{S}(\mathscr{A} \models I \wedge A \wedge \neg B(\varphi, \sigma) \Rightarrow \mathscr{A} \models B(\varphi, \tau_c(\sigma))).$$

[12] Note that only first-order formulas appear as subformulas of program formulas, and nested application of program quantifiers is not admitted.

This description of satisfaction of program formulas could be applied to arbitrary transition systems. It can be simplified further by exploiting the fact that the transition system considered here is induced by a UNITY program. We can use the syntactic description of the weakest precondition of an assignment in the usual way as known, for example, from the Hoare logic.

Definition 5.3 (*UNITY Hoare triples*). Let c be the following assignment of $\mathscr{T}$:

$$\begin{aligned} \boldsymbol{d} := \quad & t_0 \quad \text{if } E_0 \\ & \vdots \\ & \sim t_n \quad \text{if } E_n. \end{aligned}$$

With any formulas A and B, formulas $c(B)$ and $\{A\}c\{B\}$ are associated by

(1) $c(B) \stackrel{\text{def}}{\equiv} \bigwedge_{j\leqslant n}(E_j \to B[t_j/\boldsymbol{d}]) \wedge (\bigwedge_{j\leqslant n} \neg E_j \to B)$,

(2) $\{A\}c\{B\} \stackrel{\text{def}}{\equiv} \forall \boldsymbol{d}(A \to c(B))$.

The following lemma states two equivalences expressing how these UNITY Hoare triples describe the effect of assignments.

Lemma 5.4. *Let $\mathscr{A}$ be a many-sorted structure.*

(1) $\mathscr{A} \models c(B)(\varphi, \sigma) \Leftrightarrow \mathscr{A} \models B(\varphi, \tau_c(\sigma))$,

(2) $\mathscr{A} \models \{A\}c\{B\}(\varphi) \Leftrightarrow \forall \sigma \in \mathbf{S}(\mathscr{A} \models A(\varphi, \sigma) \Rightarrow \mathscr{A} \models B(\varphi, \tau_c(\sigma)))$.

The following corollary is an immediate consequence of Lemmas 5.2 and 5.4.

Corollary 5.5. *Let $\mathscr{A}$ be a many-sorted structure.*

(1) $\mathscr{T}(\mathscr{A}) \models \mathsf{stable}_{\mathscr{T}}\, I(\varphi) \Leftrightarrow \mathscr{A} \models \bigwedge_{c\in\mathscr{T}} \{I\}c\{I\}(\varphi)$,

(2) $\mathscr{T}(\mathscr{A}) \models A\, \mathsf{unless}_{\mathscr{T},I}\, B(\varphi) \Leftrightarrow \mathscr{T}(\mathscr{A}) \models \mathsf{stable}_{\mathscr{T}}\, I(\varphi) \wedge$
$\mathscr{A} \models \bigwedge_{c\in\mathscr{T}} \{I \wedge A \wedge \neg B\}c\{A \vee B\}(\varphi)$,

(3) $\mathscr{T}(\mathscr{A}) \models A\, \mathsf{ensures}_{\mathscr{T},I}\, B(\varphi) \Leftrightarrow \mathscr{T}(\mathscr{A}) \models A\, \mathsf{unless}_{\mathscr{T},I}\, B(\varphi) \wedge$
$\mathscr{A} \models \bigvee_{c\in\mathscr{T}} \{I \wedge A \wedge \neg B\}c\{B\}(\varphi)$.

5.2. *Formal proof system*

We are now prepared to define a formal proof system for deriving $\mathsf{stable}_{\mathscr{T}}$, $\mathsf{unless}_{\mathscr{T}}$, $\mathsf{ensures}_{\mathscr{T}}$, and $\mapsto_{\mathscr{T}}$ propositions of a UNITY program $\mathscr{T}$. For each program quantifier, there is exactly one proof rule. The proof rules for $\mathsf{stable}_{\mathscr{T}}$, $\mathsf{unless}_{\mathscr{T}}$, and $\mathsf{ensures}_{\mathscr{T}}$ are straightforward formalizations of the semantic definitions. They are suggested directly by Corollary 5.5. The proof rule for $\mapsto_{\mathscr{T}}$ is a formalization of our principle of transfinite induction for $\mapsto$.

With one exception, only first-order formulas occur in these proof rules. This exception is the formula expressing well-foundedness of a suitable relation $\prec$ occur-

ring in the principle of transfinite induction. Well-foundedness of a relation definable in $\mathscr{L}_0$ can be expressed by a Π^1_1-formula of the second order extension $\mathscr{L}_1$ of $\mathscr{L}_0$. In the following definition a formula serving this purpose is defined. Familiar abbreviations are used.

Definition and Lemma 5.6 (Well-foundedness and Π^1_1). (1) *Let O be a formula, s a sort, and $y{:}s \not\equiv x{:}s$ variables of $\mathscr{L}_0$. Then we define*

$$WF(\{(y,x)|O\}) \stackrel{\text{def}}{\equiv} \forall X{:}\,[s]\quad (X \neq \emptyset \rightarrow (\exists x{:}s \in X)(\forall y{:}s \in X) \neg O).$$

(2) *A formula of the second-order extension $\mathscr{L}_1$ of $\mathscr{L}_0$ is called Π^1_1 iff it is obtained from a formula containing only first-order quantifiers by prefixing universal predicate quantifiers.*

(3) *For $O, s, y{:}s$, and $x{:}s$ as in* (1) *above, $WF(\{(y,x)|O\})$ is a Π^1_1-formula of $\mathscr{L}_1$.*

Definition 5.7 (*Logic $\mathbf{UY}^{\mathscr{T}}$ of UNITY Program $\mathscr{T}$*). (1) The formal system $\mathbf{UY}^{\mathscr{T}}$ is given by the following rules:

$$\text{(stable)}\quad \frac{\bigwedge_{c \in \mathscr{T}} \{I\}c\{I\}}{\text{stable}_{\mathscr{T}}\, I}$$

$$\text{(unless)}\quad \frac{\text{stable}_{\mathscr{T}}\, I,\ \bigwedge_{c \in \mathscr{T}} \{I \wedge A \wedge \neg B\}c\{A \vee B\}}{A \text{ unless}_{\mathscr{T},I}\, B}$$

$$\text{(ensures)}\quad \frac{A \text{ unless}_{\mathscr{T},I}\, B,\ \bigvee_{c \in \mathscr{T}} \{I \wedge A \wedge \neg B\}c\{B\}}{A \text{ ensures}_{\mathscr{T},I}\, B}$$

$$(\mapsto \textit{ind})\quad \frac{WF(\{(y,x)|O\}),\ I \wedge A \rightarrow B \vee \exists x{:}s R,\ R \text{ ensures}_I\, B \vee \exists y{:}\, s(O \wedge R[y/x])}{A \mapsto_{\mathscr{T},I} B}$$

provided O is a formula of $\mathscr{L}_0$ not containing the program variables $\boldsymbol{d}$ free, $y{:}s$, $x{:}s$ are different from each other and from the program variables $\boldsymbol{d}$, and $y{:}s$, $x{:}s$ do not occur free in $I, A, B, \exists x{:}\, sR$.

(2) Let Γ be a set of formulas of $\mathscr{L}_1$, and C a program formula of $\mathscr{L}_0^{\mathscr{T}}$. A $\mathbf{UY}^{\mathscr{T}}$-*derivation* of C from Γ is a tree of formulas with formulas of Γ at it leaves and using the rules of $\mathbf{UY}^{\mathscr{T}}$ at its interior nodes. The derivability relation of $\mathbf{UY}^{\mathscr{T}}$ is defined by

$$\Gamma \vdash_{\mathbf{UY}^{\mathscr{T}}} C \stackrel{\text{def}}{\Leftrightarrow} \text{there is a } \mathbf{UY}^{\mathscr{T}}\text{-derivation of } C \text{ from } \Gamma.$$

Soundness and relative completeness of $\mathbf{UY}^{\mathscr{T}}$ are formulated with reference to the Π^1_1-theory of a structure $\mathscr{A}$, which is defined as follows.

Definition 5.8 (Π^1_1-*Theory of a structure*). Let $\mathscr{A}$ be a many-sorted structure. The Π^1_1-theory of $\mathscr{A}$ is defined by

$$\Pi^1_1\text{-Th}(\mathscr{A}) \stackrel{\text{def}}{=} \{C \mid C\, \Pi^1_1\text{-formula of } \mathscr{L}_1 \text{ and } \mathscr{A} \models C\}.$$

It is easy to see that $\mathbf{UY}^{\mathscr{T}}$ is sound w.r.t. the intended interpretation in the program $\mathscr{T}(\mathscr{A})$.

Lemma 5.9 (Soundness of $\mathbf{UY}^{\mathscr{T}}$). *Let $\mathscr{A}$ be a many-sorted structure and φ a valuation in $\mathscr{A}$.*

(1) *For each of the rules* (stable), (unless), *and* (ensures), *the conclusion of the rule is satisfied in $\mathscr{T}(\mathscr{A})$ under φ if and only if all premisses of the rule are satisfied in $\mathscr{A}$ or in $\mathscr{T}(\mathscr{A})$ under φ.*

(2) *If the premisses of the rule ($\mapsto$ ind) are satisfied in $\mathscr{A}$ or in $\mathscr{T}(\mathscr{A})$ under φ, then the conclusion is satisfied in $\mathscr{T}(\mathscr{A})$ under φ.*

Proof. (1) is an immediate consequence of Corollary 5.5. To prove (2), consider an application of rule ($\mapsto$ *ind*) with $O, I, A, B, R, y{:}s, x{:}s$ are required by the side conditions of the rule. Define

$$W \stackrel{\text{def}}{=} |\mathscr{A}|_s, \qquad \prec \stackrel{\text{def}}{=} \{(u,w)\in|\mathscr{A}|_s^2 \mid \mathscr{A}\models O(\varphi[u,w/y,x])\},$$

$$\mathbf{I} \stackrel{\text{def}}{=} \{\sigma\in\mathbf{S} \mid \mathscr{A}\models I(\varphi,\sigma)\}, \qquad \mathbf{P} \stackrel{\text{def}}{=} \{\sigma\in\mathbf{S} \mid \mathscr{A}\models A(\varphi,\sigma)\},$$

$$\mathbf{Q} \stackrel{\text{def}}{=} \{\sigma\in\mathbf{S} \mid \mathscr{A}\models B(\varphi,\sigma)\}, \qquad \mathbf{R}_w \stackrel{\text{def}}{=} \{\sigma\in\mathbf{S} \mid \mathscr{A}\models R(\varphi[w/x],\sigma)\}.$$

The hypotheses of Proposition 3.1 are satisfied with these items, since by assumption the premisses of this application of rule ($\mapsto$ *ind*) are satisfied under φ. $\mathscr{T}(\mathscr{A})\models A\mapsto_{\mathscr{T},I} B(\varphi)$ follows by Proposition 3.1 and semantic correctness of $\mapsto$ w.r.t. leads-to. □

Proposition 5.10 (Soundness of $\mathbf{UY}^{\mathscr{T}}$). *Let $\mathscr{A}$ be a many-sorted structure and C a program formula of $\mathscr{L}_0^{\mathscr{T}}$, then*

$$\Pi_1^1\text{-Th}(\mathscr{A})\vdash_{\mathbf{UY}^{\mathscr{T}}} C \;\Rightarrow\; \mathscr{T}(\mathscr{A})\models C.$$

Proof. Observe that a $\mathbf{UY}^{\mathscr{T}}$-derivation consists of at most one application of each of the rules (stable), (unless), (ensures), and ($\mapsto$ *ind*). The formulas used in this derivation are valid in $\mathscr{A}$ by hypothesis. So by Lemma 5.9, applications of the rules result in program formulas valid in $\mathscr{T}$. □

To prove relative completeness of $\mathbf{UY}^{\mathscr{T}}$, we have to make some further assumptions about the structure $\mathscr{A}$, over which the UNITY program $\mathscr{T}$ computes. These requirements are captured in the notion of an arithmetical structure, which we define here to be a model of a theory in a language of sufficient expressive power.

Definition 5.11 (*Arithmetical structure*). (1) $\mathscr{L}_0$ is called an *arithmetical language for* $\mathscr{T}$ if it contains nat as a sort with equality, an individual constant $\mathsf{O}:\mathsf{nat}$, function constants $\mathsf{S}:[\mathsf{nat}]\to\mathsf{nat}$, $+:[\mathsf{nat},\mathsf{nat}]\to\mathsf{nat}$ and $*:[\mathsf{nat},\mathsf{nat}]\to\mathsf{nat}$, a predicate constant $<$: $[\mathsf{nat},\mathsf{nat}]$, and constants for the primitive recursive functions and predicates over nat used in the proof of the Semantic Completeness Theorem 3.2.

Furthermore it has to contain function constants $p:[s_1,\ldots,s_k]\to\mathrm{nat}$ and $p_i:[\mathrm{nat}]\to s_i$ $(1\leqslant i\leqslant k)$.[13]

(2) A set of axioms in the language $\mathscr{L}_1$ is called an *arithmetical theory for* $\mathscr{T}$ if the first-order part $\mathscr{L}_0$ of $\mathscr{L}_1$ is arithmetical for $\mathscr{T}$, and if it contains standard primitive recursive defining equations or equivalences for the constants required to be in the language and furthermore the following axioms:

$$p_i(p(d_1,\ldots,d_k))=d_i \quad (1\leqslant i\leqslant k), \qquad WF(\{(y,x)\mid y<x\}).$$

(3) $\mathscr{A}$ is called an *arithmetical structure for* $\mathscr{T}$ if it is a model of an arithmetical theory for $\mathscr{T}$.

The usual requirement that functions for the coding of finite sequences and a copy of the standard model of arithmetic are first-order definable in an arithmetical structure follows from our definition. By virtue of the validity of the Π^1_1-formula $WF(\{(y,x)\mid y<x\})$ and of the standard defining equations and equivalences in an arithmetical structure $\mathscr{A}$, the following holds: The carrier $|\mathscr{A}|_{nat}$ is the set of (standard) natural numbers, the constants for functions and predicates over the natural numbers have in $\mathscr{A}$ their standard meaning, and the constants $p:[s_1,\ldots,s_k]\to\mathrm{nat}$ and $p_i:[\mathrm{nat}]\to s_i$ $(1\leqslant i\leqslant k)$ are interpreted by functions for coding states of $\mathscr{T}(\mathscr{A})$ by natural numbers.

If it were wanted, the requirement that a theory be arithmetical could be modified to demanding the presence of axioms in the reduced language containing just the constants $\mathsf{O}:\mathrm{nat}$, $\mathsf{S}:[\mathrm{nat}]\to\mathrm{nat}$, $+:[\mathrm{nat},\mathrm{nat}]\to\mathrm{nat}$, $*:[\mathrm{nat},\mathrm{nat}]\to\mathrm{nat}$, $p:[s_1,\ldots,s_k]\to\mathrm{nat}$, and $p_i:[\mathrm{nat}]\to s_i$ $(1\leqslant i\leqslant k)$. It is a well-known fact, that all the rest is then first-order definable. We have required more constants to be in the language in order to conveniently formulate a refinement of our relative completeness result, which gives information about the complexity of the auxiliary formulas occurring in a derivation of $\mathbf{UY}^{\mathscr{T}}$. The following classification of formulas according to their quantifier complexity is used to express this refinement.

Definition 5.12 (Δ^0_0,Σ^0_1). Let $\mathscr{L}_0$ be an arithmetical language for $\mathscr{T}$.

(1) A formula of $\mathscr{L}_0$ is called Δ^0_0 iff it contains only bounded quantifiers over variables of sort nat, i.e., quantifiers of the form $\forall x{:}\,\mathrm{nat}(x<t\to\cdots)$ *or* $\exists x{:}\,\mathrm{nat}(x<t\wedge\cdots)$.

(2) A formula of $\mathscr{L}_0$ is called Σ^0_1 iff it is obtained from *a* Δ^0_0-formula by prefixing first-order existential quantifiers.

Theorem 5.13 (Relative completeness of $\mathrm{UY}^{\mathscr{T}}$). *If* $\mathscr{A}$ *is an arithmetical structure, and* C *a program formula of* $\mathscr{L}^{\mathscr{T}}_0$, *then the following hold:*

(1) $\mathscr{T}(\mathscr{A})\models C \Rightarrow \Pi^1_1\text{-Th}(\mathscr{A})\vdash_{\mathrm{UY}^{\mathscr{T}}} C$.

(2) *If* $C\equiv A\mapsto_{\mathscr{T},I} B$ *and* I, A, *and* B *are* Δ^0_0*-formulas, then in the application of rule* ($\mapsto$ *ind*) *the formula* O *can be choosen to be a* Δ^0_0*-formula and* R *a* Σ^0_1*-formula.*

[13] Recall that $s_1,\ldots,s_k$ are the sorts of the program variables $d_1,\ldots,d_k$ of $\mathscr{T}$.

Proof (*sketch*). (1) If C is a stable, unless or ensures formula, then the claim is easy to verify. By Lemma 5.9, the conclusion of each of the rules (stable), (unless), or (ensures) is equivalent to the conjunction of its premisses. So if $\mathscr{T}(\mathscr{A}) \models C$, the $\mathscr{L}_0$-formulas used in the premisses are in the first-order theory of $\mathscr{A}$, and a fortiori in Π^1_1-Th($\mathscr{A}$).

For the case that C is a $\mapsto$ formula, we have to formalize the notions defined in the proof of the Semantic Completeness Theorem 3.2. Using the functions $p, p_1, \ldots, p_k$ in $\mathscr{A}$ for coding states as natural numbers, one can assume coding techniques to be applied as described at the beginning of Section 4.2. Let I, A and B be given s.t. $C \equiv A \mapsto_{\mathscr{T},I} B$ and $\mathscr{T}(\mathscr{A}) \models A \mapsto_{\mathscr{T},I} B$. Choose variables x:nat, y:nat, which are different from each other and from the program variables, and do not occur free in I, A, and B. The property of being the code of a state can be expressed by $p(p_1(x), \ldots, p_k(x)) = x$. Further one can define Δ^0_0-formulas $\leqslant^*$, *Exec*, and $\lessdot$ defining in $\mathscr{A}$ the predicates $\leqslant^*$, $Exec_{\mathscr{T}}$, and $\lessdot$ respectively. The formula *Exec* contains exactly one free variable, say x. $\leqslant^*$ and $\lessdot$ contain exactly two free variables, say y and x. By an abuse of notation we write $t \leqslant^* t'$ instead of $\leqslant^*[y, x/t, t']$, and similarly for $\lessdot$. Next one defines formulas W and O as follows:

$$W \stackrel{\text{def}}{\equiv} Exec \wedge |x|\ odd \wedge (x \neq \langle\,\rangle \rightarrow (I \wedge A)[\boldsymbol{p}((x)_0)/\boldsymbol{d}])$$
$$\wedge\ \forall i < |x|(i\ even \rightarrow \neg B[\boldsymbol{p}((x)_i)/\boldsymbol{d}]),$$
$$O \stackrel{\text{def}}{\equiv} W \wedge W[y/x] \wedge y \lessdot x.$$

Finally a formula R is defined by

$$R \stackrel{\text{def}}{\equiv} \exists z\text{: nat}(W \wedge W[z/x] \wedge z \leqslant^* x \wedge \neg z \lessdot x \wedge last(z) = p(\boldsymbol{d})).$$

Observe that W contains as free variables x and the free variables of I, A and B, except the program variables. The free variables of R are x, the free variables of I, A and B, and the program variables $\boldsymbol{d}$. With these definitions the premises of rule ($\mapsto$ *ind*) can be seen to be valid in $\mathscr{A}$ and in $\mathscr{T}$, and hence derivable in $\mathbf{UY}^{\mathscr{T}}$ from Π^1_1-Th($\mathscr{A}$). Applying rule ($\mapsto$ *ind*) gets us where we want.

(2) Under the additional hypotheses of (2), the formula O defined above is clearly Δ^0_0, and likewise the formula R is Σ^0_1. □

6. Conclusions

What has been achieved? We have fulfilled our aim of finding a formalized finitary proof system for $\mapsto$ which is correct and complete for leads-to relative to a Π^1_1-complete set of formulas.

But then, since leads-to itself is also a Π^1_1-notion, why not adopt the 'proof rule' saying outright that $A \mapsto_{\mathscr{T},I} B$ is provable if A leads-to$_I$ B is valid in $\mathscr{T}$? The answer is the same as given by Apt and Plotkin in their discussion in [3, p. 759]. While on a theoretical level such a 'proof rule' is just as complex as our system, in a practical

sense it would be useless. After all one wants a proof system precisely in order to *get to know* the validity of A leads-to$_I$ B. So the above 'proof rule' is a tautology from a practical point of view. In contrast, as Chandy and Misra's work shows, proving leads-to relations with the help of the notions stable, unless, ensures, and transfinite induction for $\mapsto$ is a promising method. So our proof system is much more practical than the purported 'proof rule' above.

Still the proof system presented here is not suggested to be an adequate formal system for UNITY logic. There are mainly two deficiencies.

Firstly, what about the dozens of rules used in [5] and shown to be of practical value? They are proved in [5] in the sense that a semantic relation $\mapsto$ similar to the one defined in Definition 2.4(2) is closed under these rules. By appeal to our Syntactic Completeness Theorem 5.13, we can conclude that our proof system is also closed under (suitable refinements of) these rules. But this draws on a Π^1_1-oracle, which in practice is certainly not available. Any formal system with a recursive set of axioms can only generate a Σ^0_1-subset of the Π^1_1-set we are after. Concretely speaking, we will have available only those hypotheses of well foundedness in rule ($\mapsto$ *ind*), which are derivable in any given formal theory we choose to use for the predicate logic part of $\mathbf{UY}^{\mathcal{F}}$. And as $\mathbf{UY}^{\mathcal{F}}$ stands, we have no reason to believe that such a recursively enumerable fragment of it will also be closed under derived rules like those of [5]. To arrive at a practically useful formal proof system for UNITY logic, more rules have to be added as primitives in order to compensate for the lack of a Π^1_1-oracle. The choice of a suitable axiomatic basis guaranteeing useful closure properties of recursively enumerable subsystems of UNITY logic is an open task.

Secondly, $\mathbf{UY}^{\mathcal{F}}$ talks only about one specific UNITY program. To make use of the compositionality properties of the UNITY operators, one needs a framework in which one can talk about several UNITY programs and their combinations. There will have to be further proof rules dealing with combinations of programs, for example, rules expressing the theorems in [24, Section 5]. The design of a suitable language and of rules serving this purpose in an open task, too.

Related work. The most comprehensive and systematic work to date addressing the foundations of UNITY is a recent paper by Jutla and Rao [13]. This is discussed below in the context of other foundational work dealing directly with UNITY. We begin our survey with earlier work in the theory of nondeterministic programming languages restricting ourselves to aspects directly related to our results. For general background we refer to [3] for countable nondeterminism and to [8] for fairness in finite nondeterminism as comprehensive sources containing more references.

Principles similar to our transfinite induction for $\mapsto$ have been studied in work on fair termination of nondeterministic programs. In particular the 'Method M' of Lehmann et al. [16] for proving so-called *impartial convergence* is comparable. Francez has in [8, Section 2.2] adapted this to a method – called 'state directed choice of helpful directions' – for proving *unconditionally-fair termination* of programs in his language of guarded commands. Lehmann et al. give a direct proof of the completeness of their method, which also was adapted by Francez to his framework. Compared

to these proofs, our completeness proof is much simpler. Due to this, formalization of our proof is straightforward, and we end up with an assertion language which is simpler than the languages suggested by other authors for similar purposes. For example, Francez treats the question of syntactic completeness of his methods in [8, Section 6.4]. He chooses a version of $\mathbf{L}_\mu$ (see [8, Section 6.2]) as assertion language for a relatively complete syntactic proof rule. $\mathbf{L}_\mu$ is an extension of Hitchcock and Park's μ-calculus [11] by a sort for the ordinals, constants for all recursive ordinals, and the ordering relation between them. The μ-calculus, in turn, extends first-order predicate logic by fixpoint operators.

The same kind of assertion language is also used by Apt and Plotkin in their study of countable nondeterminism [3]. Apt and Plotkin investigate a programming language incorporating countable nondeterminism via so-called *random assignment*. They suggest a Hoare-like proof system for the total correctness of programs of this language. Also they have a completeness theorem for their proof system. More precisely, they prove completeness relative to validity of formulas of the form $A \rightarrow B$, where A and B are the so-called positive formulas of their language. Positive formulas are equivalent to Π^1_1-formulas. Their completeness property thus uses an oracle, which is properly more complex than Π^1_1 (see [3, Section 5.3]).

Another approach to countable nondeterminism is by Mascari and Venturini Zilli in the framework of *algorithmic logic*. In [17] they develop a logic ALNA (Algorithmic Logic for *while*-programs with Nondeterministic Assignment), which is correct and complete. The assertion language of ALNA is a powerful infinitary language, in which well-foundedness of the computation tree of a program can be expressed.

In the context of pure CCS Darondeau and Yoccoz [7] study the logical complexity of various notions of testing equivalence. The infinitary testing equivalence $\approx$ – the most complex of these notions – is Π^1_2-complete. It is pointed out that in the framework of Girard's Π^1_2-logic complete proof systems for their notions exist, however none are described explicitly.

Some interesting work addressing foundational questions deals directly with UNITY. In contrast to our paper, this work is mostly in the tradition of the *predicate transformer approach* to the semantics of programs. In some of this work, also the question of finding a complete system of 'proof rules' has been studied. However, we have nowhere found a presentation of a finitary formal system for the logic of UNITY. Completeness is mostly dealt with only in the sense of semantic completeness.

Some authors prove completeness by appealing to a known completeness result for some system of temporal logic. Gerth and Pnueli [9] treat UNITY by comparison with a closely related class of programs in the language of guarded commands, called SLP (*single location programs*). As to the computational power they observe that all partial recursive (deterministic) functions are computable by SLP-programs. They propose a programming logic for these programs obtained within Manna and Pnueli's temporal logic framework. The main proof rule E for $\diamond$ (*eventually*) is similar to our principle of transfinite induction for $\mapsto$. Sanders [24] clarifies the role of the Substitution Axiom by suggesting refinements of the original UNITY notions taken

up here. Furthermore she proves a semantic completeness result for $\mapsto$ by reduction to completeness of Gerth and Pnueli's temporal logic for SLP-programs. The question of finding a suitable assertion language is touched, and the language $\mathbf{L}_\mu$ of [8, Section 6.2] is conjectured to be of sufficient expressive power.

Pachl [21] is the first to stress the infinitary character of the UNITY 'proof rules' for $\mapsto$. In order to point out that a finitary proof system for leads-to cannot be complete in the absolute sense, he reduces the Π^0_1-complete complement of *Post's Correspondence Problem* to a schema of leads-to properties. Furthermore he gives a direct proof of semantic completeness of $\mapsto$ w.r.t. leads-to without using temporal logic. This proof is very short and elegant and has been published in [22]. However, as Pachl points out, his proof is not constructive, and (at least to me) it does not suggest a syntactic proof rule, which could be hoped to be relatively complete. Essentially the same nonconstructive completeness proof has been discovered earlier by Cohen and is presented in a generalized framework in [6]. Cohen works in a more abstract setting of transition systems over a complete atomic boolean algebra. The proof system suggested by him is infinitary. His results are not directly relevant for our work, since he extends the program model of UNITY in essential ways by allowing *infinite* programs containing *nondeterministic* transitions.

The paper [13] by Jutla and Rao contains a thorough investigation of the semantics of fair UNITY-style programs and of proof rules for them by means of the predicate transformer approach. In some respects it goes beyond our concerns in this paper. The main contribution of Jutla and Rao is the design of a methodology for finding correct and relatively complete proof rules for progress properties associated with *various* fairness conditions, of which UNITY's unconditional fairness is just one example. Also, reviewing the classical results for Hoare logic, they discuss in detail the problem of the expressiveness of a suitable assertion language. Their methodology goes from a branching time temporal logic (CTL*) characterization to a fixpoint characterization of the fairness notion in question, and from this via a predicate transformer for the associated progress property to an equivalent description by means of a relation. This relation is a 'proof rule' in the sense that it is generated by infinitary closure conditions from a more elementary notion (analogous to the generation of $\mapsto$ from ensures). A completeness proof thus comes as a by-product of applying this methodology, however, the issue is clearly one of *semantic* completeness. This completeness proof is direct,[14] but nonconstructive. Only fixpoint operators are needed to express the predicates occurring in the proof. Furthermore a second, constructive proof of semantic completeness is given for the case of unconditional fairness. This constructive version, however, uses ordinals in addition to the fixpoint operators. While it seems possible to extract from it a *formal* proof system and a proof of relative completeness in the *syntactic* sense, this has not been done in the paper. Clearly the

[14] In an earlier version of some of these results (see [12]), semantic completeness is proved by appeal to temporal logic.

formal language obtained in this way would be a version of L_μ. The methodology of Jutla and Rao seems rather useful as a heuristic to find good proof rules. But the completeness proof obtained by their method leads to unnecessary requirements on the expressiveness of an assertion language for a relatively complete formal system. As our results show, it is overkill to have two nonelementary features in the language: the μ-operator *and* ordinals. Our results also raise doubts about the appropriateness of using the 'weakest predicate **P** s.t. **P leads-to Q**' in order to find the auxiliary predicates needed in a complete proof system. In particular, our analysis of the halting problem shows that this predicate transformer in the worst case transforms recursive sets into Π^1_1-sets, whereas in applying our transfinite induction principle for $\mapsto$ we can get by with Σ^0_1-sets as auxiliaries.

References

[1] K.R. Apt, Ten years of Hoare's logic: A survey – Part I, *ACM Trans. Programming Languages and Systems* **3** (1981) 431–483.

[2] K.R. Apt and E.-R. Olderog, Proof rules and transformations dealing with fairness, *Sci. Comput. Programming* **3** (1983) 65–100.

[3] K.R. Apt and G.D. Plotkin, Countable nondeterminism and random assignment, *J. Assoc. Comput. Mach.* **33** (1986) 724–767.

[4] A.K. Chandra, Computable nondeterministic functions, in: *Proc 19th Ann. Symp. on Foundations of Computer Science* (IEEE, New York, 1978) 127–131.

[5] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation* (Addison-Wesley, Reading, MA, 1988).

[6] E. Cohen, A complete, compositional proof system for infinite, unconditionally fair transition systems, Tech. Report 92-24, Dept. of Computer Sciences, Univ. of Texas at Austin, 1992.

[7] P. Darondeau and S. Yoccoz, Proof systems for infinite behaviours, *Inform. and Comput.* **99** (1992) 178–191.

[8] N. Francez, *Fairness*, Texts and Monographs in Computer Science (Springer, New York, 1986).

[9] R. Gerth and A. Pnueli, Rooting UNITY, *ACM SIGSOFT Engineering Notes*, 14(3) (1989) 11–19 extended abstract.

[10] D. Harel, *First-Order Dynamic Logic*, Lecture Notes in Computer Science, Vol. 68 (Springer, Berlin, 1979).

[11] P. Hitchcock and D. Park, Induction rules and termination proofs, in: M. Nivat, ed., *Automata, Languages and Programming* (North-Holland, Amsterdam, 1973).

[12] C.S. Jutla, E. Knapp and J.R. Rao, A predicate transformer approach to semantics of parallel programs, in: *Proc. 8th Ann. ACM Symp. on Principles of Distributed Computing* (ACM, New York, 1989) 249–263.

[13] C.S. Jutla and J.R. Rao, On a fixpoint semantics and the design of proof rules for fair parallel programs, Tech. Report 92-23, Dept. of Computer Sciences, Univ. of Texas at Austin, 1992.

[14] E.R. Knapp, A predicate transformer for progress, *Inform. Process. Lett.* **33**(6) February (1989/90) 323–330.

[15] D. Kozen and J. Tiuryn, Logics of programs, in: J. van Leeuwen, ed., *Formal Models and Semantics, Handbook of Theoretical Computer Science*, Vol. B chapter 14 (Elsevier, Amsterdam, 1990) 787–840.

[16] D. Lehmann, A. Pnueli and J. Stavi, Impartiality, justice and fairness: the ethics of concurrent termination, in: S. Even and O. Kariv, eds., *Automata, Languages and Programming*, Lecture Notes in Computer Science Vol. 115 (Springer, Berlin, 1981) 264–277.

[17] G. Mascari and M. Venturini Zilli, While-programs with nondeterministic assignments and the logic ALNA, *Theoret. Comput. Sci.* **40** (1985) 211–235.

[18] J. Misra, Soundness of the Substitution Axiom, Notes on UNITY 14-90, Univ. of Texas at Austin, 1990.
[19] Y.N. Moschovakis, *Elementary Induction on Abstract Structures* (North-Holland, Amsterdam, 1974).
[20] P. Odifreddi, *Classical Recursion Theory. The Theory of Functions and Sets of Natural Numbers*, Studies in Logic and the Foundations of Mathematics, Vol. 125 (North-Holland, Amsterdam, 1989).
[21] J. Pachl, Three Definitions of *leads-to* for UNITY, Notes on UNITY 23-90, Univ. of Texas at Austin, 1990.
[22] J. Pachl, A simple proof of a completeness result for *leads-to* in the UNITY Logic, *Inform. Process. Lett.* **41** (1) (1992) 35–38.
[23] J.R. Rao, On a Notion of Completeness for the Leads-to, Notes on UNITY 24-90, Univ. of Texas at Austin, 1991.
[24] B.A. Sanders, Eliminating the Substitution Axiom from UNITY Logic, *Formal Aspects Comput.* **3** (1991) 189–205.