Journal of Universal Computer Science, vol. 14, no. 6 (2008), 801-844
submitted: 18/10/07, accepted: 26/2/08, appeared: 28/3/08 © J.UCS

The Computable Multi-Functions on Multi-represented Sets are Closed under Programming

Klaus Weihrauch
(University of Hagen, Germany
klaus.weihrauch@fernuni-hagen.de)

Abstract: In the representation approach to computable analysis (TTE) [Grz55, KW85, Wei00], abstract data like rational numbers, real numbers, compact sets or continuous real functions are represented by finite or infinite sequences $(\Sigma^*, \Sigma^\omega)$ of symbols, which serve as concrete names. A function on abstract data is called computable, if it can be realized by a computable function on names. It is the purpose of this article to justify and generalize methods which are already used informally in computable analysis for proving computability. As a simple formalization of informal programming we consider flowcharts with indirect addressing. Using the fact that every computable function on Σ^ω can be generated by a monotone and computable function on Σ^* we prove that the computable functions on Σ^ω are closed under flowchart programming. We introduce generalized multi-representations, where names can be from general sets, and define realization of multi-functions by multi-functions. We prove that the function computed by a flowchart over realized functions is realized by the function computed by the corresponding flowchart over realizing functions. As a consequence, data from abstract sets on which computability is well-understood can be used for writing realizing flowcharts of computable functions. In particular, the computable multi-functions on multi-represented sets are closed under flowchart programming. These results allow us to avoid the "use of 0s and 1s" in programming to a large extent and to think in terms of abstract data like real numbers or continuous real functions. Finally we generalize effective exponentiation to multi-functions on multi-represented sets and study two different kinds of λ-abstraction. The results allow simpler and more formalized proofs in computable analysis.
Key Words: computable analysis, multi-functions, multi-representations, realization, flowcharts, λ-abstraction
Category: F.0

1 Introduction

By the Church/Turing Thesis a partial word function $f :\subseteq (\Sigma^*)^k \to \Sigma^*$ is computable by a digital computer, iff it can be computed by a Turing machine [HU79], which is the mathematical model of computation introduced by A.Turing in 1936 [Tur36]. This concept of computability can be extended to countable sets of "abstract" data like natural or rational numbers or finite graphs. For this purpose abstract data are encoded by "concrete" words $w \in \Sigma^*$ and a function on abstract data is called computable, if it can be realized by a computable word function on codes.

Computability on the set of real numbers cannot be introduced in this way since the supply Σ^* of (finite) names is only countable. Various models have been proposed for studying aspects of computability on the real numbers. For discussion of models see, for example, [SHT99], [Wei00, § 9], [TZ04]. This article is based on the representation approach (Type-2 theory of effectivity, TTE) to computable analysis [KW85, Wei00], which generalizes a definition of computable real functions introduced by Grzegorczyk and Lacombe [Grz55, Lac55, Grz57].

In TTE, abstract data like real numbers, compact sets or continuous real functions are represented by *infinite* sequences $p \in \Sigma^\omega$ of symbols from a finite alphabet Σ which serve as concrete names. A function on abstract data is called computable, if it can be realized by a computable function on names. Concrete computations on infinite sequences of symbols can be defined, for instance, by Type-2 machines which are Turing machines with infinite input and output tapes performing infinite computations. For example, for the real numbers the "Cauchy representation" $\rho_C :\subseteq \Sigma^\omega \to \mathbb{R}$ [Wei00] induces the Grzegorczyk/Lacombe computability (GL-computability) of real functions. Addition, division, the exponential function and many other real functions are GL-computable. Infinite sequences of natural numbers ($\mathbb{B} = \mathbb{N}^\mathbb{N}$, Baire space) can be used as names instead of infinite sequences of symbols (Σ^ω, Cantor space) equivalently.

The representation approach as a foundation of computability in analysis has been criticized repeatedly, for example in [BCSS98]: " ... the Turing model ... with its dependence on 0s and 1s is fundamentally inadequate for giving a foundation to the theory of modern scientific computation where most of the algorithms – with origins in Newton, Euler, Gauss, et al. – are *real number algorithms*." and after some further comments: " These reasonings give some justification for taking as a model for scientific computation a machine model that accepts real numbers as inputs." Furthermore, in more advanced applications of TTE where computability of higher level operators is proved [ZW03] the use of Σ^ω as set of names is cumbersome and may hide the main ideas which should be expressed more abstractly.

For an algorithmic foundation of numerical analysis in [BSS89] generalized Turing machines have been suggested which explicitly operate on real numbers instead of symbols from a finite alphabet with vectors of real numbers as inputs and outputs, real number assignments "$x \leftarrow 1$" and "$x \leftarrow y$ op z" for the algebraic operations op $\in \{+, -, \cdot, /\}$, and real number branchings "$x < y$?". Although this model looks very concrete and expresses the way most numerical mathematicians are presumably thinking, the "BSS-machines" are unrealistic for the following reason: Computers cannot handle exact real numbers but (usually) operate on floating point numbers instead. Since these computer REALs are not even closed under the arithmetic operations, rounding errors occur repeatedly during computations. Such errors are usually tolerable for the basic algebraic operations, since they are continuous but may be disastrous if they cause false branchings. Notice that branchings such as $< : \mathbb{R}\times\mathbb{R} \to \{0, 1\}$ are not continuous.

The BSS model is neither sound nor complete for the computable real functions [Wei00, Section 9.7].

In computer science higher level programming languages are introduced in order to avoid thinking in terms of concrete computations on 0s and 1s at the machine level. Apart from BSS-machines there are several approaches to formalize programming in analysis. As a modification of the BSS-machines the "feasible real-RAMs" [BH98] have "approximate" tests [BC90], which are multi-valued, instead of ordinary tests and a limit operator. Feasible real-RAMs operate directly on the real numbers. They can be translated to the realistic Type-2 machines. Since every computable real function can be obtained in this way, this abstract model is sound and complete. Moreover, feasible real-RAMs have realistic computational complexity. Another approach is Brattka's generalization of the definition of the μ-recursive functions [Bra96, Bra03]. His "defining terms" considered as a programming language can be translated to Type-2 machines computing the real functions (w.r.t. the standard representation of the real numbers) and more generally computable functions on admissibly represented metric spaces. The language is sound and complete for the computable multi-functions (see also [Her99]). The real numbers and other spaces can be embedded in interval domains [Sco70]. Languages for exact real number computation can be developed from this idea [Esc96, ES99, EE01, EMR07, DG99, CDG06]. As an abstract model of computation in [TZ04] Tucker and Zucker choose the "while"-array programming language [TZ99, TZ00], extended with a nondeterministic "countable choice" assignment, called the $\mathbf{WhileCC}^*$ model. Using this model they introduce the concept of *approximable many-valued computation* on metric algebras. For metric algebras with an *effective representation* α, $\mathbf{WhileCC}^*$ approximability implies computability in α and under certain reasonable conditions the converse is true. Their article generalizes and extends results from [Ste99].

In terms of [TZ04] Theorem 30, a main result of this paper, can be described as follows. Instead of the $\mathbf{WhileCC}^*$ model we use flowcharts with indirect addressing which are essentially equivalent. In particular, we do not provide a language syntactically, and we assume that computability on finite and infinite sequences of symbols as well as on the natural numbers is already given. The metric partial algebra is replaced by a (finite) set of multi-represented sets and relatively computable (or continuous) functions which can be used in the flowchart. It is shown that the multi-function computed by the flowchart is again relatively computable (continuous). This corresponds to the soundness result in [TZ04], where, however, soundness is considered only for the restriction of the computed function to the computable points of the metric algebras with respect to canonical numberings α (called "representations" in [TZ04]). This soundness theorem is an immediate consequence of Theorems 15 and 23 (provided the difference between **WhileCC*** and flowcharts is ignored).

In this article, which extends the preliminary version [Wei05], we add a toolbox to the existing framework of TTE. In Section 2 we recapitulate how continuous (computable) functions on Σ^ω can be generated by monotone (monotone computable) functions on Σ^* [Wei00]. In Section 3 we introduce composition and restriction for multi-functions. In Section 4 we define flowcharts with indirect addressing and their semantics. In Section 5 we consider flowcharts operating on Σ^ω with continuous functions. We prove that the function computed by such a flowchart is generated by the word function computed by the flowchart, which is obtained from the first one by replacing all the functions on Σ^ω by generating word functions. Consequently, if only computable functions on Σ^ω are used, the flowchart computes a computable function. In Section 6 we introduce generalized multi-representations where "generalized" means that arbitrary sets can be used as sets of names and "multi" means that many objects may have a common name (see the explanations in Section 6). Generalized representations allow us the use of simpler but still abstract data as names instead of sequences of symbols. This generalizes [Bla00] where domains are allowed as sets of names. In Section 7 we prove for generalized multi-representations and realizing as well as realized multi-functions: the function computed by a flowchart over realized functions is realized by the function computed by the corresponding flowchart over realizing functions. In Section 8 we consider computability. As the most important result we conclude that the relatively computable multi-functions are closed under flowchart programming. This generalizes results in [BH98] (feasible real RAMs), in [Wei00] (closure under primitive recursion) and in [Bra96, Bra03]. By the other main result, instead of using "concrete" names from Σ^ω we may use data from more abstract sets on which computability is well-understood for writing realizing flowcharts of computable functions. Both results allow us to get rid of the 0s and 1s in programming and to think in terms of abstract data like real numbers or continuous real functions. In Section 9 we generalize effective exponentiation to multi-representations and generalize the important type conversion theorem [Wei00, Theorem 3.3.15].

The main motivation for this article came from the wish to make practical work in computable analysis easier. The results should simplify (or justify the current practice of) many proofs of computability in advanced applications. Although it does not intend to design a programming language this article can be considered as a step towards a higher level programming language for computable analysis.

2 Computability on Sequences of Symbols

The computable functions on finite and infinite sequences of symbols can be defined by generalized Turing machines. In this section we introduce some notations and extend two results from Chapter 2 in [Wei00], namely, we show that continuous functions can be generated by monotone-constant or monotone word functions, and that computable functions can be generated by monotone-constant or monotone computable word functions. Furthermore, the composition of functions is generated by the composition of generators.

Throughout this paper we use the terminology from [Wei00]. Let Σ be a finite alphabet such that $\{0,1\}\subseteq\Sigma$. Let Σ^* be the set of finite words over Σ with the empty word ϵ, and let $\Sigma^\omega := \{a_0a_1\ldots \mid a_i \in \Sigma\}$ be the set of all infinite sequences over Σ. By $|w|$ we denote the length of the word $w \in \Sigma^*$, and by $\sqsubseteq$ the prefix relation on $\Sigma^* \cup \Sigma^\omega$. We extend the prefix relation to vectors by $(x_1,\ldots,x_k) \sqsubseteq (y_1,\ldots,y_k) \iff (\forall i)x_i \sqsubseteq y_i$. For $p = a_0a_1a_2\ldots \in \Sigma^\omega$ let $p^{<k} := a_0\ldots a_{k-1}$ be the prefix of $p \in \Sigma^\omega$ of length k. By $\langle\, ,\, \rangle$ we denote various computable standard injective tupling functions, see Definition 2.1.7 in [Wei00]. In particular, for $i,j \in \mathbb{N} := \{0,1,2,\ldots\}$, let $\langle i,j\rangle := (i+j)\cdot(i+j+1)/2+j$ be the Cantor pairing function. Using the "wrapping function" $\iota : \Sigma^* \to \Sigma^*$, $\iota(a_1\ldots a_k) := 110a_10\ldots0a_k011$ we define $\langle x,y\rangle := \iota(x)\iota(y)$ and $\langle x,p\rangle := \langle p,x\rangle := \iota(x)p$ for $x,y \in \Sigma^*$ and $p \in \Sigma^\omega$. For $p,q \in \Sigma^\omega$, let $\langle p,q\rangle := (p(0)q(0)p(1)q(1)\ldots)$, and for $p_0,p_1,p_2,\ldots \in \Sigma^\omega$, define $\langle p_0,p_1,p_2,\ldots\rangle \in \Sigma^\omega$ by $\langle p_0,p_1,p_2,\ldots\rangle(\langle i,j\rangle) := p_i(j)$.

On Σ^* we consider the discrete topology (i.e. every set $X\subseteq\Sigma^*$ of words is open). On Σ^ω we consider the Cantor topology defined by the basis $\{w\Sigma^\omega \mid w \in \Sigma^*\}$ (i.e. every set $w\Sigma^\omega := \{q \in \Sigma^\omega \mid w \sqsubseteq q\}$ is open, and every open set is a union of such sets). On Cartesian products we consider the product topologies.

A Type-2 machine M is a multi-tape Turing machine with k input tapes (for some $k \geq 0$) finitely many work tapes and a single one-way (!) output tape together w ith a type specification $(Y_1,\ldots,Y_k \to Y_0)$, $Y_i \in \{\Sigma^*,\Sigma^\omega\}$ (Figure 1). The function $f_M :\subseteq Y_1 \times\ldots\times Y_k \to Y_0$ computed by the Type-2 machine M is defined as follows:

Case $\mathbf{Y_0 = \Sigma^*}$: $f_M(p_1,\ldots,p_k) = w$, iff M halts on input $(p_1,\ldots,p_k)$ with $w \in \Sigma^*$ on the output tape;

Case $\mathbf{Y_0 = \Sigma^\omega}$: $f_M(p_1,\ldots,p_k) = p_0$, iff M computes forever on input $(p_1,\ldots,p_k)$ and writes $p_0 \in \Sigma^\omega$ on the output tape.

As usual we write $f(y)\downarrow$ for $y \in \mathrm{dom}(f)$ and $f(y)\uparrow$ for $y \notin \mathrm{dom}(f)$. It is known that every function $f :\subseteq Y_1 \times\ldots\times Y_k \to Y_0$ computable in this way is continuous, and that its domain is an r.e. open set for $Y_0 = \Sigma^*$ and a G_δ-set (i.e. a countable intersection of open sets) in the Kleene class Π_2 for $Y_0 = \Sigma^\omega$. Since later on we will not consider natural domains the following (somewhat unusual) terminology is convenient.

Definition 1 *(Turing computable, computable).* We call a function $f :\subseteq Y_1 \times \ldots\times Y_k \to Y_0$ *Turing computable*, iff $f = f_M$ for some Type-2 machine M, and we call it *computable*, if it has a Turing computable extension.

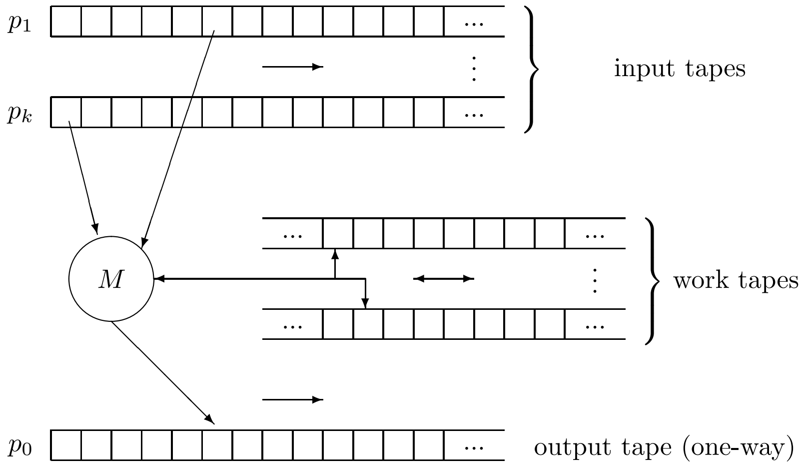

Figure 1: A Type-2 machine

Continuous Type-2 functions can be approximated by word functions. In the following we generalize [Wei00, Lemma 2.1.11]. For technical reasons we use “momotone-constant” functions instead of monotone-constant functions and partial monotone functions instead of total monotone functions.

Definition 2. Call a function $h :\subseteq (\Sigma^*)^k \to \Sigma^*$ *monotone-constant*, iff

$$(h(y)\downarrow \ \text{and} \ \ y \sqsubseteq y') \implies (h(y')\downarrow \ \text{and} \ \ h(y) = h(y')),$$

and *monotone*, iff

$$(h(y)\downarrow \ \text{and} \ \ y \sqsubseteq y') \implies (h(y')\downarrow \ \text{and} \ \ h(y) \sqsubseteq h(y')).$$

For monotone-constant functions g define $T_*(g) :\subseteq (\Sigma^\omega)^k \to \Sigma^*$, and for monotone functions h define $T_\omega(h) :\subseteq (\Sigma^\omega)^k \to \Sigma^\omega$ by

$$T_*(g)(x) = w :\iff (\exists y \in (\Sigma^*)^k)\,(y \sqsubseteq x \ \text{ and } \ g(y) = w),$$
$$T_\omega(h)(x) = q :\iff q = \sup\nolimits_{\sqsubseteq}\{h(y) \mid y \sqsubseteq x \ \text{ and } \ h(y) \ \text{ exists}\}.$$

Notice that $T_*(g)$ and $T_\omega(h)$ are well-defined by the “generating functions” g and h, respectively. By Lemma 3 Turing computable functions $f :\subseteq (\Sigma^\omega)^k \to \Sigma^*$ or $f' :\subseteq (\Sigma^\omega)^k \to \Sigma^\omega$ can be generated by computable word functions $g :\subseteq (\Sigma^*)^k \to \Sigma^*$ which are monotone-constant or monotone, respectively. We include the continuous versions.

Lemma 3. *1. A function $f :\subseteq (\Sigma^\omega)^k \to \Sigma^*$ is continuous with open domain, iff $f = T_*(g)$ for some monotone-constant function $g :\subseteq (\Sigma^*)^k \to \Sigma^*$.*

2. *A function $f :\subseteq (\Sigma^\omega)^k \to \Sigma^*$ is Turing computable, iff $f = T_*(g)$ for some Turing computable monotone-constant function $g :\subseteq (\Sigma^*)^k \to \Sigma^*$.*

3. *A function $f :\subseteq (\Sigma^\omega)^k \to \Sigma^\omega$ is continuous with G_δ-domain, iff $f = T_\omega(h)$ for some monotone function $h :\subseteq (\Sigma^*)^k \to \Sigma^*$.*

4. *A function $f :\subseteq (\Sigma^\omega)^k \to \Sigma^\omega$ is Turing computable, iff $f = T_\omega(h)$ for some Turing computable monotone function $h :\subseteq (\Sigma^*)^k \to \Sigma^*$.*

For a proof modify the proofs of Lemma 2.1.11 and Theorem 2.3.7 in [Wei00] appropriately. The proof shows how a Type-2 machine can be converted to a Turing machine of a generating function and conversely. The Turing computable functions are "almost" closed under composition.

Lemma 4 [Wei00]. *For $k, n \in \mathbb{N}$ and $X_1, \ldots, X_k, Y_1, \ldots, Y_n, Z \in \{\Sigma^*, \Sigma^\omega\}$ and Turing computable functions $g_i :\subseteq X_1 \times \ldots \times X_k \to Y_i$ and $f :\subseteq Y_1 \times \ldots \times Y_n \to Z$ ($i = 1, \ldots, n$), the composition $f \circ (g_1, \ldots, g_n) :\subseteq X_1 \times \ldots \times X_k \to Z$*

1. *is Turing computable, if $Z = \Sigma^\omega$ or $Y_i = \Sigma^*$ for all i,*

2. *has a Turing computable extension $h :\subseteq X_1 \times \ldots \times X_k \to Z$ such that*

$$\mathrm{dom}(h) \cap \mathrm{dom}(g_1, \ldots, g_n) = \mathrm{dom}(f \circ (g_1, \ldots, g_n)) \ ,$$

if $Z = \Sigma^*$ and $Y_i = \Sigma^\omega$ for some i.

Proof: Although the proof is easy and can be found in [Wei00]), we sketch it here for the case $k = n = 1$. Let M and N be Type-2 machines computing f and g_1, respectively.

If $Y_1 = \Sigma^*$ let L be a Type-2 machine which on input x first runs the machine N to compute $g_1(x) \in \Sigma^*$ and then runs the machine M on input $g_1(x)$.

If $Y_1 = \Sigma^\omega$, let L be a machine which runs M and N alternately using the output tape of N as the input tape of M. It runs N on input x until it writes one symbol and interrupts, then it runs one step of M, then it continues the computation of N until it writes one symbol and interrupts, then it runs one further step of M, and so on. For $Z = \Sigma^\omega$ the machine L computes $f \circ g_1$, and for $Z = \Sigma^*$ the machine L computes an extension h of $f \circ g_1$ such that $\mathrm{dom}(h) \cap \mathrm{dom}(g_1) = \mathrm{dom}(f \circ g_1)$. □

By the following lemma, composition of functions on Σ^ω can be obtained by composing generating functions. We denote the restriction of $f :\subseteq X \to Y$ to $Z \subseteq X$ by $f|_Z :\subseteq X \to Y$.

Lemma 5. *Let $g_1, \ldots, g_n :\subseteq (\Sigma^*)^k \to \Sigma^*$ be monotone, let $f :\subseteq (\Sigma^*)^n \to \Sigma^*$ be monotone-constant and let $h :\subseteq (\Sigma^*)^n \to \Sigma^*$ be monotone. Then*

$$T_*(f) \circ \big(T_\omega(g_1), \ldots, T_\omega(g_n)\big) = T_*\big(f \circ (g_1, \ldots, g_n)\big)\big|_{\bigcap_i \mathrm{dom}(T_\omega(g_i))}, \tag{1}$$

$$T_\omega(h) \circ \big(T_\omega(g_1), \ldots, T_\omega(g_n)\big) = T_\omega\big(h \circ (g_1, \ldots, g_n)\big)\big|_{\bigcap_i \mathrm{dom}(T_\omega(g_i))}. \tag{2}$$

Proof: For convenience we consider $n = 1$. Suppose, $T_\omega(g)(p)$ exists.
(1) For $p \in \Sigma^\omega$ and $x \in \Sigma^*$,

$$\begin{aligned} T_*(f) \circ T_\omega(g)(p) = x &\iff (\exists y \sqsubseteq T_\omega(g)(p))\, f(y) = x \\ &\iff (\exists y, z)\,(z \sqsubseteq p\,,\ y \sqsubseteq g(z) \ \text{ and } \ f(y) = x) \\ &\iff (\exists z \sqsubseteq p)\ f \circ g(z) = x \\ &\iff \ T_*(f \circ g)(p) = x\,. \end{aligned}$$

(2) For $p \in \Sigma^\omega$ and $v \in \Sigma^*$,

$$\begin{aligned} v \sqsubseteq T_\omega(h) \circ T_\omega(g)(p) &\iff (\exists\, w \in \Sigma^*)\big(w \sqsubseteq T_\omega(g)(p) \ \text{ and } \ v \sqsubseteq h(w)\big) \\ &\iff (\exists\, w, u \in \Sigma^*)\big(u \sqsubseteq p\,,\ w \sqsubseteq g(u) \ \text{ and } \ v \sqsubseteq h(w)\big) \\ &\iff (\exists\, u \in \Sigma^*)\big(u \sqsubseteq p \ \text{ and } \ v \sqsubseteq h \circ g(u)\big) \\ &\iff v \sqsubseteq T_\omega(h \circ g)(p)\,. \end{aligned}$$

□

3 Multi-Functions

Many problems in computable analysis can be solved not by computable functions, but only by computable multi-functions [Luc77]. In such cases the result of a computation may depend on the used name, different names of the same object may give different results. In this section we introduce multi-functions, which are also called *correspondences*. For further details see [KT84, Wei00]. We distinguish two kinds of composition and we generalize the "restriction" relation from (single-valued) functions to multi-functions.

A *multi-valued function*, *multi-function* for short, from A to B is a triple $f = (A, B, \mathrm{R}_f)$ such that $\mathrm{R}_f \subseteq A \times B$ (the *graph* of f). We will denote it by $f : A \rightrightarrows B$. For $a \in A$ let $f(a) := \{b \in B \mid (a, b) \in R_f\}$. For $X \subseteq A$ let $f[X] := \{b \in B \mid (\exists a \in X)(a, b) \in R_f\}$, $\mathrm{dom}(f) := \{a \in A \mid f(a) \neq \emptyset\}$, and $\mathrm{range}(f) := f[A]$. If, for every $a \in A$, $f(a)$ contains at most one element, f is a usual partial function denoted by $f :\subseteq A \to B$.

In applications multi-functions can express various ideas, for example, nondeterminism, randomness, uncertainty, reachability or set-valued functions. In this article we have two kinds of usage, which are distinguished also formally by the operation of composition. In one of the applications we have in mind, for a multi-function $f : A \rightrightarrows B$, $f(a)$ is interpreted as the set of all results which are "acceptable" on input $a \in A$. Any concrete computation, a realization of f, will produce on input $a \in \mathrm{dom}(f)$ some element $b \in f(a)$, but often there is no method to select a specific one.

Example 1 (multi-functions). 1. $f : \mathbb{R} \rightrightarrows \mathbb{Q},\ \ f(x) := \{r \in \mathbb{Q} \mid x < r\}$,

2. $g : \mathbb{R} \times \mathbb{R} \rightrightarrows \mathbb{N},\ \ g(x_1, x_2) := \{i \in \mathbb{N} \mid x_i > 0\}$,

3. $\leq_k : \mathbb{R} \times \mathbb{R} \rightrightarrows \mathbb{N},\ \ \leq_k (x, y) := \begin{cases} \{0\} & \text{if } \ x < y \\ \{0,1\} & \text{if } \ y \leq x \leq y + 2^{-k} \\ \{1\} & \text{if } \ y + 2^{-k} < x\,, \end{cases}$

4. $h : C(\mathbb{R}) \rightrightarrows \mathbb{R},\ \ h(f) := \{x \in \mathbb{R} \mid f(x) = 0\}$.

In TTE with respect to the canonical representations the function f (find some rational upper bound), the function g (find some i such that $x_i > 0$) and $\leq_k$ (approximate branching) are computable but cannot be replaced by computable single-valued functions while the multi-function h (zero finding) is not even computable [Wei00]. Functions like g can be applied in Gaussian elimination, where indices i, j must be determined such that $a_{ij} \neq 0$. Approximate branching can be used in Newton iteration where some n must be determined such that after n iterations the error is less than a given bound. More generally, every branching in a naive real number algorithm gives rise to a multi-function, since characteristic functions $\mathrm{cf}_A : \mathbb{R}^n \to \mathbb{N}$ (for non-trivial $A \subseteq \mathbb{R}^n$) cannot be realized exactly (Cor. 4.3.16 in [Wei00]).

Notice that a multi-function $f : A \rightrightarrows B$ is well-defined by the values $f[\{a\}] = f(a) \subseteq B$ for all $a \in A$. Therefore, a multi-function from A to B could also be defined as a usual (single-valued) function $f : A \to 2^B$. We will write $f(a) = \uparrow$, if $a \notin \mathrm{dom}(f)$, that is $f[\{a\}] = \emptyset$. For multi-functions $f_i : A \rightrightarrows B_i$ $(i = 1 \ldots, k)$ define the juxtaposition $(f_1, \ldots, f_k) : A \rightrightarrows B_1 \times \ldots \times B_k$ by

$$(f_1, \ldots, f_k)(a) := f_1(a) \times \ldots \times f_k(a)\,.$$

And for multi-functions $f_i : A_i \rightrightarrows B_i$ $(i = 1 \ldots, k)$ define the product $(f_1 \times \ldots \times f_k) : A_1 \times \ldots \times A_k \rightrightarrows B_1 \times \ldots \times B_k$ by

$$(f_1 \times \ldots \times f_k)(a_1, \ldots, a_k) := f_1(a_1) \times \ldots \times f_k(a_k)\,.$$

In this article we consider two kinds of composition of multi-functions.

Definition 6 *(composition).* For $f : A \rightrightarrows B$ and $g : B \rightrightarrows C$ define

1. $g \odot f : A \rightrightarrows C$ by $\ g \odot f(a) \ := \ g[f(a)]$,

2. $g \circ f : A \rightrightarrows C$ by $g \circ f(a) := g[f(a)]$ for all $a \in \mathrm{dom}(g \circ f)$, where

$$a \in \mathrm{dom}(g \circ f) :\Longleftrightarrow a \in \mathrm{dom}(f) \text{ and } f(a) \subseteq \mathrm{dom}(g).$$

Notice that $a \in \mathrm{dom}(g \odot f) \iff a \in \mathrm{dom}(f)$ and $f(a) \cap \mathrm{dom}(g) \neq \emptyset$. For single-valued f, $g \odot f = g \circ f$. The composition $\odot$ is the usual composition of binary relations and can be used, for example, for modelling reachability or nondeterminism. The composition $\circ$ is appropriate for the multi-functions in Example 1. We generalize the concept of restriction/extension from (single-valued) functions to multi-functions as follows.

Definition 7. For multi-functions $f, g : A \rightrightarrows B$,

$$f \trianglelefteq g :\Longleftrightarrow \big(\mathrm{dom}(f) \subseteq \mathrm{dom}(g) \text{ and } (\forall a \in \mathrm{dom}(f))\, g(a) \subseteq f(a)\big)$$

(f "restricts" g, g "extends" f).

For single-valued functions, $f \trianglelefteq g$ means that f restricts g in the usual meaning. For multi-functions $f \trianglelefteq g$ can be interpreted as g "tightens" or "improves" f. For $a \in A$, "$f(a) = B$" is better or sharper than "$f(a) =\uparrow$", since "every value $b \in B$ is acceptable" is better than "no $b \in B$ is acceptable". And for $X, Y \subseteq B$, $f(a) = X$ is sharper or tighter than $f(a) = Y$, if $\emptyset \neq X \subseteq Y$. Notice that $f \trianglelefteq g$ is not related to graph inclusion. Examples: Let $R_f = \{(0,0),(0,1)\}$ and $R_g = \{(0,0),(1,1)\}$. Then $f \trianglelefteq g$ (f restricts g, g is sharper than f) but neither $R_f \subseteq R_g$ nor $R_g \subseteq R_f$. Let $R_f = \{(0,0)\}$ and $R_g = \{(0,0),(0,1),(1,1)\}$. Then $R_f \subseteq R_g$ but neither $f \trianglelefteq g$ nor $g \trianglelefteq f$.

Lemma 8. *1. The total single-valued functions are the maximal elements in the $\trianglelefteq$-order.*

2. The compositions $\odot$ and $\circ$ are associative.

3. The composition $\circ$ is monotone w.r.t. extension:
$g \circ f \trianglelefteq g' \circ f'$ *if* $f \trianglelefteq f'$ *and* $g \trianglelefteq g'$.

4. The composition $\odot$ is monotone w.r.t. graph inclusion:
$R_{g \odot f} \subseteq R_{g' \odot f'}$ *if* $R_f \subseteq R_{f'}$ *and* $R_g \subseteq R_{g'}$.

4 Flowcharts with Indirect Addressing

The Turing computable functions $f :\subseteq (\Sigma^*)^k \to \Sigma^*$ are closed under composition. More generally, they are closed under "flowchart computation", that is, the function computed by a flowchart composed of Turing computable word functions is Turing computable. In Section 5 we prove that also the computable functions on Σ^ω are closed under flowchart computation. In this section we define flowcharts and their operational semantics on arbitrary data types. Since direct addressing fixes the number of variables which can be used in a flowchart we consider indirect addressing. For convenience addresses (and Boolean values) are words in Σ^*.

Although dated, flowcharts are still used in computable analysis for proving computability informally. Since in this article we are not interested in defining a programming language, we do not define GOTO programs or WHILE programs syntactically but introduce flowcharts mathematically like Turing machines or finite automata and thus presumably make it easier for the non-expert to understand the formalism and meaning of the theorems. For our purpose it is convenient to define (uninterpreted) *flowchart schemes* and then flowcharts by assigning types to the variables and typed functions to the function symbols.

Definition 9 *(flowchart scheme).* Let $\mathrm{Var}\subseteq\Sigma^*$ be a recursive set (set of variables). A flowchart scheme F is given by

1. a finite set Q of labels;
2. an initial label $l_{in} \in Q$ and a final label $l_{fin} \in Q$ $(l_{in} \neq l_{fin})$;
3. a finite set Fun of function names, each $f \in \mathrm{Fun}$ with fixed arity $\mu(f) \in \mathbb{N}$;
4. a mapping Stmt assigning to each label $l \in Q \setminus \{l_{fin}\}$ a statement
$$\mathrm{Stmt}(l) = (u := f(u_1, \ldots, u_{\mu(f)}),\ l') \tag{3}$$
or
$$\mathrm{Stmt}(l) = (\text{if } u \text{ then } l', l'') \tag{4}$$
such that $l', l'' \in Q$, $f \in \mathrm{Fun}$ and $u, u_1, \ldots, u_{\mu(f)} \in \mathrm{Var}$;
5. a vector $(v_1, \ldots, v_m)$ of pairwise different input variables $v_i \in \mathrm{Var}$;
6. the output variable $v_0 \in \mathrm{Var}$.

Flowcharts operate on structures $\tau = (X_\tau, \ldots)$ (where X_τ is a set), which we will call *types* in this article. For convenience, if in a context a type τ is used like a set, then it means the set X_τ. For example, $x \in \tau$ means $x \in X_\tau$ and $f : \tau_1 \times \ldots \times \tau_k \rightrightarrows \tau$ means a multi-function $f : X_{\tau_1} \times \ldots \times X_{\tau_k} \rightrightarrows X_\tau$.

In the following we will consider fixed types **Pointer** and **Bool** such that $X_{\mathbf{Pointer}} = \mathrm{Var}\subseteq\Sigma^*$ and $X_{\mathbf{Bool}} = \mathrm{Bool} = \{0,1\}\subseteq\Sigma^*$.

Definition 10 *(interpretation, flowchart).* An interpretation I of the flowchart scheme F is given by a finite set $\mathcal{T}^I$ of types and a function assigning to each variable $v \in \mathrm{Var}$ a type $I(v) \in \mathcal{T}^I$ and to each function name $f \in \mathrm{Fun}$ a multi-function f^I such that

1. $\{\mathbf{Pointer}, \mathbf{Bool}\} \in \mathcal{T}^I$;
2. $I(v_1), \ldots, I(v_m) \notin \{\mathbf{Pointer}, \mathbf{Bool}\}$;
3. $f^I : \tau_1 \times \ldots \times \tau_{\mu(f)} \rightrightarrows \tau$ for types $\tau_1, \ldots, \tau_{\mu(f)} \in \mathcal{T}^I \setminus \{\mathbf{Pointer}, \mathbf{Bool}\}$ and $\tau \in \mathcal{T}^I$;
4. the function $v \mapsto I(v)$ is computable.

We call the pair (F, I) a flowchart.

In (4) "computable" means computable w.r.t. some (arbitrary) injective notation ν of the finite set $\mathcal{T}^I$. We define the multi-function f_F^I computed by the flowchart (F, I) by operational semantics. A computation is a sequence $(l_i, \sigma_i)_i$ of configurations where the l_i are labels and the σ_i are states. A state assigns a value of correct type (or the undefined $\uparrow$) to every variable. The first two lines in Figure 3 shows an example for values $I(v)$

Definition 11 *(semantics).* Let (F, I) be a flowchart.
A state is a mapping σ on the set Var of variables such that

$$\sigma(v) \in I(v) \cup \{\uparrow\}\,.$$

Let State^I be the set of states. Let $Q \times \mathrm{State}^I$ be the set of configurations. Define the next-relation $\overset{I}{\vdash}$ on the set of configurations as follows:

$$(l, \sigma) \overset{I}{\vdash} (\bar{l}, \bar{\sigma}) :\Longleftrightarrow \text{ 1. or 2.} \tag{5}$$

1. **Assignment:** There are variables $u, u_1, \ldots, u_k \in \mathrm{Var}$ of type **Pointer** and a function symbol $f \in \mathrm{Fun}$ such that

$$- \mathrm{Stmt}(l) = \quad (u := f(u_1, \ldots, u_k),\ \bar{l})\,, \tag{6}$$
$$- f^I : I \circ \sigma(u_1) \times \ldots \times I \circ \sigma(u_k) \rightrightarrows I \circ \sigma(u)\,, \tag{7}$$
$$- \bar{\sigma} \circ \sigma(u) \in f^I(\sigma \circ \sigma(u_1), \ldots, \sigma \circ \sigma(u_k))\,, \tag{8}$$
$$- \bar{\sigma}(v) = \sigma(v) \quad \text{for} \quad v \neq \sigma(u)\,. \tag{9}$$

2. **Branching:** There are a variable $u \in \mathrm{Var}$ of type **Pointer** and labels $l', l'' \in Q$ such that

$$- \mathrm{Stmt}(l) = \ (\text{if } u \text{ then } l', l'') \tag{10}$$
$$- I(\sigma(u)) = \mathbf{Bool}\,, \tag{11}$$
$$- \begin{cases} \sigma \circ \sigma(u) = \mathtt{1} \text{ and } (\bar{l}, \bar{\sigma}) = (l', \sigma) \\ \quad \text{or} \\ \sigma \circ \sigma(u) = \mathtt{0} \text{ and } (\bar{l}, \bar{\sigma}) = (l'', \sigma)\,. \end{cases} \tag{12}$$

Consider $x = (x_1, \ldots, x_m) \in I(v_1) \times \ldots \times I(v_m)$.
An "x-computation" is a finite or infinite sequence $(l_0, \sigma_0), (l_1, \sigma_1), \ldots$ of configurations such that:

$$- \; l_0 = l_{in}\,; \tag{13}$$

$$- \begin{cases} \sigma_0(v) = v \quad \text{if} \quad I(v) = \textbf{Pointer}\,, \\ \sigma_0(v_1) = x_1, \; \ldots, \; \sigma_0(v_m) = x_m, \\ \sigma_0(v) = \uparrow \;\; \text{otherwise}\,; \end{cases} \tag{14}$$

$$- \; (l_{i-1}, \sigma_{i-1}) \stackrel{I}{\longmapsto} (l_i, \sigma_i) \quad \text{for} \quad i = 1, 2, \ldots\,. \tag{15}$$

A computation is maximal, if it is infinite or it is finite and its last element has no $\stackrel{I}{\longmapsto}$ –successor. A computation is acceptable, if it is finite, and its last configuration is (l_{fin}, σ) for some state σ such that $\sigma(v_0)$ exists.

The function $f^I_F : I(v_1) \times \ldots \times I(v_m) \rightrightarrows I(v_0)$ computed by the flowchart (F, I) is defined as follows:

$$f^I_F(x) = \begin{cases} \{y \mid y = \sigma_n(v_0) \;\; \text{for the last configuration} \;\; (l_n, \sigma_n) \\ \qquad\qquad \text{of some maximal } x\text{-computation}\} \\ \qquad \text{if every maximal } x\text{-computation is acceptable,} \\ \emptyset \quad \text{otherwise.} \end{cases} \tag{16}$$

Notice that in case of assignment implicitly

- $l \neq l_{fin}$ in (5),
- $\sigma(u), \sigma(u_1), \ldots, \sigma(u_k) \in \text{Var}$ exist,
- $\bar{\sigma} \circ \sigma(u), \; \sigma \circ \sigma(u_1), \ldots, \sigma \circ \sigma(u_k)$ exist,
- $f^I(\sigma \circ \sigma(u_1), \ldots, \sigma \circ \sigma(u_k))$ exists.

and in case of branching implicitly

- $l \neq l_{fin}$ in (5),
- $\sigma(u) \in \text{Var}$ exists,
- $\sigma \circ \sigma(u) \in \text{Bool}$ exists.

By (16), $f^I_F(x) \neq \emptyset$ (exists) if, and only if, *every* maximal x-computation is acceptable. This condition generalizes the definition of $\text{dom}(g \circ f)$ (Definition 6) for multi-functions. Since $v \mapsto I(v)$ is Turing computable, the type conditions (7) and (11) can be checked computably during runtime. By Definition 10.3, functions cannot be applied to variables or Boolean values, i.e. addresses or Boolean values cannot be used to compute new data. This restriction, however, is imposed merely for simplifying our proofs by reducing the number of cases. It is inessential since in applications the set $\mathcal{T}^I$ of types may contain further types behaving like **Bool** and **Pointer** as ordinary data which can be used without limitations. Of course, for non-trivial flowcharts we need non-trivial functions $f^I : \tau_1 \times \ldots \times \tau_{\mu(f)} \rightrightarrows \tau$ for types $\tau_1, \ldots, \tau_{\mu(f)} \in \mathcal{T}^I \setminus \{\mathbf{Pointer}, \mathbf{Bool}\}$ and $\tau \in \{\mathbf{Pointer}, \mathbf{Bool}\}$ from "abstract" types to the fixed concrete data types **Pointer** or **Bool**. By (14) at the beginning $\sigma_0(v) = v$, hence $\sigma_0 \circ \sigma_0(v) = v$. In order to get indirect access to other variables w the flowchart scheme must have assignments of the form $(u := h_w(), l)$, which must be interpreted by $I(h)() := w \in \text{Var}$.

Example 2 (acceptable computation). Consider Heron's method to compute the squareroot of a real number $a > 0$ with small error given by $e \in \mathbb{Q}$: $x_0 := a$, $x_{n+1} := (x_n + a/x_n)/2)$, halt, if $|x_n^2 - a| \leq e$. The following informally written flowchart F_0 with *direct* addressing solves the problem. The input (a, e) is in (v_1, v_2), the output is in v_0. Define $h : \mathbb{R} \to \mathbb{R}$, $g :\subseteq \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and $f : \mathbb{R} \times \mathbb{R} \times \mathbb{Q} \rightrightarrows \text{Bool}$ by $h(x) := x$, $g(x, a) := (x + x/a)/2$ and $f(x, a, e) = \{0\}$ for $|x^2 - a| \geq e$, $f(x, a, e) = \{1\}$ for $|x^2 - a| \leq e/2$ and $f(x, a, e) = \{0, 1\}$ otherwise.

$$
\begin{aligned}
&0 : v_0 := h(v_1), 1;\\
&1 : v_0 := g(v_0, v_1), 2;\\
&2 : v_3 := f(v_0, v_1, v_2), 3;\\
&3 : \text{if } v_3 \text{ then } 4, 1.
\end{aligned}
$$

The statement at Label 2 means "compute some $b \in f(v_0, v_1, v_2)$". We convert this flowchart to a flowchart (F, I) with indirect addressing for the same purpose. Let F be the following flowchart scheme:

$$
\begin{array}{ll}
5 : u_0 := \overline{c}_0(), 6 & 0 : u_0 := \overline{h}(u_1), 1;\\
6 : u_1 := \overline{c}_1(), 7 & 1 : u_0 := \overline{g}(u_0, u_1), 2;\\
7 : u_2 := \overline{c}_2(), 8 & 2 : u_3 := \overline{f}(u_0, u_1, u_2), 3;\\
8 : u_3 := \overline{c}_3(), 0 & 3 : \text{if } u_3 \text{ then } 4, 1.
\end{array}
$$

Thus, we have labels $Q = \{0, 1, \ldots, 8\}$, $l_{in} = 5$, $l_{fin} = 4$, variables $\{u_0, \ldots, u_3, v_0, \ldots, v_3\} \in \text{Var}$ and function names $\overline{c}_0, \ldots, \overline{c}_3, \overline{f}, \overline{g}, \overline{h}$. We interpret the flowchart as follows: The types $I(v)$ of the variables can be seen in Figure 2. The interpretations of the function symbols are $I(\overline{c}_0)() := v_0, \ldots, I(\overline{c}_3)() := v_3$, $I(\overline{f}) := f$, $I(\overline{g}) := g$, and $I(\overline{h}) := h$.

In the flowchart (F, I) at label $5 + i$, the pointer u_i is initialized to the variable v_i $(i = 0, 1, 2, 3)$. Then an indirect access in state σ to u_i gives the value $\sigma \circ \sigma(u_i) = \sigma(v_i)$, that is, the same value as a direct access to v_i. Therfore, the flowchart (F, I) computes the same function as F_0. As an example, Figure 2 shows a computation with input $(2, 1/3)$. Since $1/6 < |(3/2)^2 - 2| < 1/3$, $\sigma_7(v_3) := 1$ leads to another (shorter) acceptable computation.

label	Variable v	u_0	u_1	u_2	u_3	v_0	v_1	v_2	v_3	$\dots$
	$I(v)$	**Poin.**	**Poin.**	**Poin.**	**Poin.**	**R**	**R**	**Q**	**Bool**	$\dots$
5	$\sigma_0(v)$	u_0	u_1	u_2	u_3	$\uparrow$	2	1/3	$\uparrow$	$\dots$
6	$\sigma_1(v)$	v_0	u_1	u_2	u_3	$\uparrow$	2	1/3	$\uparrow$	$\dots$
$\dots$										
0	$\sigma_4(v)$	v_0	v_1	v_2	v_3	$\uparrow$	2	1/3	$\uparrow$	$\dots$
1	$\sigma_5(v)$	v_0	v_1	v_2	v_3	2	2	1/3	$\uparrow$	$\dots$
2	$\sigma_6(v)$	v_0	v_1	v_2	v_3	3/2	2	1/3	$\uparrow$	$\dots$
3	$\sigma_7(v)$	v_0	v_1	v_2	v_3	3/2	2	1/3	0	$\dots$
1	$\sigma_8(v)$	v_0	v_1	v_2	v_3	3/2	2	1/3	0	$\dots$
2	$\sigma_9(v)$	v_0	v_1	v_2	v_3	3/2	2	1/3	0	$\dots$
3	$\sigma_{10}(v)$	v_0	v_1	v_2	v_3	17/12	2	1/3	1	$\dots$
4	$\sigma_{11}(v)$	v_0	v_1	v_2	v_3	17/12	2	1/3	1	$\dots$

Figure 2: An acceptable computation for input $(2, 1/3)$.

The following variant (F', J) of the flowchart (F, I) makes proper use of indirect addressing for storing the successive contents of register v_0 in registers $w_i := 110^i$. We introduce new variables and extend the interpretation I to J as follows: $J(u_4) := J(u_5) :=$ **Pointer**, $J(110^i) := \mathbf{R}$, $s^J : \text{Var} \to \text{Var}$, $s^J(v) := v0$, $d_1^J() := 11 \in \text{Var}$ $d_2^J() := u_4 \in \text{Var}$.

$$
\begin{array}{rlrl}
5: & u_0 := \overline{c}_0(), 6 & 0: & u_0 := \overline{h}(u_1), 1 \\
6: & u_1 := \overline{c}_1(), 7 & 1: & u_0 := \overline{g}(u_0, u_1), 12 \\
7: & u_2 := \overline{c}_2(), 8 & 12: & u_4 := h(u_0), 13 \\
8: & u_3 := \overline{c}_3(), 9 & 13: & u_5 := s(u_5), 2 \\
9: & u_4 := d_1(), 10 & 2: & u_3 := \overline{f}(u_0, u_1, u_2), 3 \\
10: & u_5 := d_2(), 0 & 3: & \text{if } u_3 \text{ then } 4, 1.
\end{array}
$$

Since u_5 points to u_4, Statement 13 changes the variable w in u_4 to the new value $w0$. Obviously, during a computaion of (F, J) the intermediate values $\sigma_i(v_0)$ (in Figure 2 the values in the column "v_0") are stored successively in the registers 11, 110, 1100, etc. and could be used in a succeding computation.

In order to simplify the framework we have considered only indirect addressing. But Example 2 shows how direct addressing can be simulated by indirect addressing. The definitions and theorems below can be generalized to flowcharts with indirect and direct addressing.

Definition 12 *(direct and indirect addressing).* Definitions 9 and 11 can be generalized to flocharts with direct and indirect addressing as follows. Instead of variables v use "marked" variables (v, s), where $s = d$ means direct addressing and $s = i$ means indirect addressing. Definition 11.1 must be adjusted appropriately.

Lemma 13. *Let F be flowchart scheme with direct and indirect addressing and let I be an interpretation of F. From F construct a flowchart scheme F' as follows: For each variable v in F add a new variable v', insert the assignment "$v' := c_v()$"at the beginning of F, and in all statements of F replace (v, d) by v' and (v, i) by v. Extend the interpretation I to I' by $I'(v') :=$ **Pointer** and $I(c_v)() := v$ for all v in F.*

Then $f^I_F = f^{I'}_{F'}$.

Proof: The statement is obvious, since after initializing the values $\sigma(v')$ to v they remain unchanged forever and every indirect access to v' is like a direct access to v since $\sigma \circ \sigma(v') = \sigma(v)$. □

If every function name is interpreted by a single-valued function, then for every input x there is at most one maximal x-computation, since the element relation "$\in$" in (8) can be replaced by equality "$=$". In this case also the function f^I_F is single-valued.

5 Computations on Σ^* Generate Computations on Σ^ω

By Lemma 4 the computable functions on Σ^ω are closed under composition, that is, $g \circ f$ is computable if f and g are computable. Although a Type-2 machine computing $g \circ f$ on input p can never finish computing the intermediate value $f(p)$, we may abstractly think as if it first computes $q := f(p) \in \Sigma^\omega$ and then $g(p) \in \Sigma^\omega$.

In this section we prove that the computable functions on Σ^ω are closed even under "flowchart programming". If we have a flowchart F composed of computable functions on Σ^ω then the function f_F computed by the flowchart is computable. Although a Type-2 machine computing f_F will never compute any intermediate value completely, we nevertheless can abstractly think as if there were a machine which computed one by one intermediate values from Σ^ω according to the rules given by the flowchart.

By Lemma 3, every computable function f on Σ^ω can be generated by a computable monotone word function h, $f = T_\omega(h)$. By Lemma 5, an extension of the composition of generated functions is generated by the composition of generating functions: $T_\omega(h_2 \circ h_1)$ extends $T_\omega(h_2) \circ T_\omega(h_1)$.

In this section we generalize this property from composition to flowcharts. Roughly speaking, let (F, J) be a flowchart composed of monotone word functions h and let (F, I) be the flowchart obtained from (F, J) by substituting $T_\omega(h)$ for h everywhere, then $T_\omega(f_F^J)$ extends f_F^I. (More precisely, the flowcharts will contain also monotone-constant functions for computing addresses and Boolean values).

In this section we consider fixed types $\boldsymbol{\Sigma^*}$ and $\boldsymbol{\Sigma^\omega}$ such that $X_{\boldsymbol{\Sigma^*}} = \Sigma^*$ and $X_{\boldsymbol{\Sigma^\omega}} = \Sigma^\omega$.

Lemma 14. *Let F be a flowchart scheme. Let I and J be interpretations such that for each $f \in \mathrm{Fun}$, f^I and f^J are single-valued and (1) to (3):*

1. $\mathcal{T}^I = \{\boldsymbol{\Sigma^\omega}, \mathbf{Pointer}, \mathbf{Bool}\}$, $\mathcal{T}^J = \{\boldsymbol{\Sigma^*}, \mathbf{Pointer}, \mathbf{Bool}\}$ *and*
$$J(v) = \begin{cases} I(v) \text{ if } I(v) \in \{\mathbf{Pointer}, \mathbf{Bool}\} \\ \boldsymbol{\Sigma^*} \text{ if } I(v) = \boldsymbol{\Sigma^\omega}\,; \end{cases}$$
2. *for $Y \in \{\mathbf{Pointer}, \mathbf{Bool}\}$, if $f^I :\subseteq (\boldsymbol{\Sigma^\omega})^k \to Y$, then $f^J :\subseteq (\boldsymbol{\Sigma^*})^k \to Y$ is monotone-constant such that $T_*(f^J)$ extends f^I;*
3. *if $f^I :\subseteq (\boldsymbol{\Sigma^\omega})^k \to \boldsymbol{\Sigma^\omega}$, then $f^J :\subseteq (\boldsymbol{\Sigma^*})^k \to \boldsymbol{\Sigma^*}$ is monotone such that $T_\omega(f^J)$ extends f^I.*

Then:

4. *If $f_F^I :\subseteq (\boldsymbol{\Sigma^\omega})^m \to Y$, $Y \in \{\mathbf{Pointer}, \mathbf{Bool}\}$, then the function $f_F^J :\subseteq (\boldsymbol{\Sigma^*})^m \to Y$ is monotone-constant and $T_*(f_F^J)$ extends f_F^I.*
5. *If $f_F^I :\subseteq (\boldsymbol{\Sigma^\omega})^m \to \boldsymbol{\Sigma^\omega}$, then the function $f_F^J :\subseteq (\boldsymbol{\Sigma^*})^m \to \boldsymbol{\Sigma^*}$ is monotone and $T_\omega(f_F^J)$ extends f_F^I.*

Proof: See Appendix A. □

We call a function to **Pointer** or **Bool** Turing computable or monotone-constant, if it is Turing computable or monotone-constant, respectively, as a function to Σ^*. Remember that by Definition 1 we call a function on Σ^* or Σ^ω computable, if it has a Turing computable extension. By the next theorem the computable functions on Σ^ω are closed under flowchart programming.

Theorem 15. *Let (F, I) be a flowchart such that $\mathcal{T}^I = \{\boldsymbol{\Sigma^\omega}, \mathbf{Pointer}, \mathbf{Bool}\}$, $I(v_0) = \boldsymbol{\Sigma^\omega}$ and f^I is computable for every function name f occurring in F. Then the function $f_F^I :\subseteq (\boldsymbol{\Sigma^\omega})^m \to \boldsymbol{\Sigma^\omega}$ is computable.*

Proof: By Lemma 3, for each f there is a monotone-constant Turing computable word function h_f such that $T_*(h_f)$ extends f^I if $f :\subseteq (\Sigma^\omega)^m \to \Sigma^*$, and a monotone Turing computable word function h_f such that $T_\omega(h_f)$ extends f^I if $f :\subseteq (\Sigma^\omega)^m \to \Sigma^\omega$. Let J be the corresponding interpretation of F such that $f^J = h_f$ for each f according to Lemma 14. Then f^J_F is monotone and $T_\omega(f^J_F)$ extends f^I_F. It is known from computability theory that the function f^J_F is Turing computable. (Remember that type checking during runtime is Turing computable.) By Lemma 3, $T_\omega(f^J_F)$ is Turing computable. Therefore, f^I_F has a Turing computable extension. □

The results in this section can be proved for flowcharts with direct addressing accordingly. Such flowcharts and their semantics can be defined without recourse to basic computability.

6 Realization for Multi-Representations

In TTE, the representation approach to computable analysis, abstract objects are "encoded" or realized by "names" $w \in \Sigma^*$ or $p \in \Sigma^\omega$, and computable functions on sets of abstract objects are realized by computable functions on names. So far mainly single-valued naming systems $\gamma :\subseteq U \to X$, $U \subseteq \{\Sigma^*, \Sigma^\omega\}$, have been considered [Wei00]. However, it has turned out that in some applications multi-valued naming systems are useful or needed (multi-representation of continuous partial functions without fixing their domains, multi-representation of the set of quasi-compact subsets of $\boldsymbol{R}_<$, which has no representation because its cardinality is bigger than that of the continuum [Wei93, Col05, GW05]). Multi-representations have been investigated in some detail in [Sch03, Wei05]. In the following we continue these studies. Since we also want to use already represented objects as names of more general ones we introduce *generalized multi-representations*. Examples are the *Domain representations* [Bla97, Bla00, BSHT02], where the standard set Σ^ω of names is replaced by Scott-Ershov domains.

Definition 16. 1. A generalized multi-representation is a multi-function $\gamma : U \rightrightarrows X$ such that range$(\gamma) = X$.

2. If $U \in \{\Sigma^*, \Sigma^\omega\}$, γ is called a multi-representation.

(Notice that in [Wei00] the term "representation" is reserved for functions $\delta :\subseteq \Sigma^\omega \to X$ and "notation" for functions $\gamma :\subseteq \Sigma^* \to X$.) Examples of representations are standard notations $\nu_\mathbb{N} :\subseteq \Sigma^* \to \mathbb{N}$ and $\nu_\mathbb{Q} :\subseteq \Sigma^* \to \mathbb{Q}$ and the Cauchy representation $\rho :\subseteq \Sigma^\omega \to \mathbb{R}$ of the real numbers [Wei00]. In [GW05] multi-representations of partial continuous functions and of quasi-compact sets are used. A multi-representation can be considered as a naming system for the points of a set X where each name can encode many points. If $x \in \delta(u)$ then in general the information given by u is not sufficient to identify the point x. A multi-representation can be interpreted also as a naming system of an *attribute* on M. However, we will use a multi-representation $\gamma : U \rightrightarrows X$ not like the single-valued representation $\bar\gamma : \subseteq U \to A$ of the set $A \subseteq 2^X$ of subsets of X such that $\bar\gamma(u) = \gamma[\{u\}]$. The difference becomes clear from the definitions of reducibility and of induced computability.

First we generalize the concept of realization [Wei00] from single-valued to multi-valued representations, and from single-valued functions on Σ^ω and Σ^* as realizations [Wei05] to arbitrary multi-functions.

Definition 17. For generalized multi-representations $\beta : U \rightrightarrows X$ and $\gamma : V \rightrightarrows Y$ and multi-functions $f : X \rightrightarrows Y$ and $r : U \rightrightarrows V$, r is a (β, γ)-realization of f, iff for all $u \in U$ and $x \in X$,

$$x \in \beta(u) \cap \mathrm{dom}(f) \Longrightarrow \big(r(u) \neq \emptyset \text{ and } \forall v \in r(u).f(x) \cap \gamma(v) \neq \emptyset\big)\,. \tag{17}$$

If β is a product, we will usually write

$$(\beta_1, \ldots, \beta_k, \gamma)\text{-realization instead of } ((\beta_1 \times \ldots \times \beta_k), \gamma)\text{-realization.} \tag{18}$$

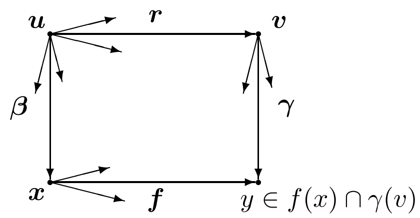

Figure 3: Each $v \in r(u)$ is a name of some $y \in f(x)$, if u is a name of $x \in \mathrm{dom}(f)$.

Whenever u is a β-name of $x \in \mathrm{dom}(f)$, then every $v \in r(u)$ is a γ-name of some $y \in f(x)$ (see Figure 3). (18) is in accordance with the definition of $(\gamma_1, \ldots, \gamma_k, \delta)$-realization introduced in [Wei00, Wei05]. The property

$$r(u) \neq \emptyset \text{ and } \forall v \in r(u).f(x) \cap \gamma(v) \neq \emptyset$$

in (17) generalizes earlier definitions. For single-valued realizing functions r it is equivalent to

$$f(x) \cap \gamma \circ r(u) \neq \emptyset \qquad \text{[Wei05]} \tag{19}$$

$$\gamma \circ r(u) \in f(x) \quad \text{if} \quad f : X \rightrightarrows Y \quad \text{and} \quad \gamma : \subseteq V \to Y \quad \text{[Wei87, Wei00]}, \tag{20}$$

$$f(x) \in \gamma \circ r(u) \quad \text{if} \quad f : \subseteq X \to Y \quad \text{and} \quad \gamma : V \rightrightarrows Y \quad \text{[Wei93, Sch03]}, \tag{21}$$

$$f(x) = \gamma \circ r(u) \quad \text{if} \quad f : \subseteq X \to Y \quad \text{and} \quad \gamma : \subseteq V \to Y \quad \text{[KW85].} \tag{22}$$

Condition (20) has been used in [Wei87, Wei00] for defining computability of multi-functions w.r.t. single-valued representations. For example, the multi-function f in Example 1 “for $x \in \mathbb{R}$ find some $r \in \mathbb{Q}$ which is slightly greater” is computable w.r.t. the standard representations of $\mathbb{R}$ and $\mathbb{Q}$ [Wei00].

In [Sch03, Page 47 (2.10)] for single-valued realizations the condition $f(x) \subseteq \gamma \circ r(u)$ is used instead of $f(x) \cap \gamma \circ r(u) \neq \emptyset$ (19). For single-valued γ, however, this condition does not reduce to the useful definition (20) and forces f to be single-valued. In particular by this definition, for no representations of $\mathbb{R}$ and $\mathbb{Q}$ the multi-function f in Example 1 is computable (or continuous).

For our generalized concept of restriction and extension for multi-functions (Definition 7) we obtain as expected:

Lemma 18. *If r is a (β, γ)-realization of f, then r' is a (β, γ)-realization of f' for every extension r' of r and every restriction f' of f.*

Proof: Suppose (17) is true for $u \in U$ and $x \in X$. Assume $x \in \beta(u) \cap \mathrm{dom}(f')$. Since $f' \trianglelefteq f$, $x \in \beta(u) \cap \mathrm{dom}(f)$. By (17) $r(u) \neq \emptyset$ and $\forall v \in r(u). f(x) \cap \gamma(v) \neq \emptyset$. Since $r \trianglelefteq r'$, $r'(u) \neq \emptyset$ and $\forall v \in r'(u). f(x) \cap \gamma(v) \neq \emptyset$. Since $x \in \mathrm{dom}(f')$, $f(x) \subseteq f'(x)$ and therefore, $\forall v \in r'(u). f'(x) \cap \gamma(v) \neq \emptyset$. □

For technical reasons we prove the following simple lemma.

Lemma 19 *(realization of tupling). For generalized multi-representations $\beta : U \rightrightarrows X$ and $\gamma_i : V_i \rightrightarrows Y_i$ $(i = 1, \dots, k)$ let $r_i : U \rightrightarrows V_i$ be a (β, γ_i)-realization of $f_i : X \rightrightarrows Y_i$. Then $(r_1, \dots, r_k)$ is a $(\beta, (\gamma_1 \times \dots \times \gamma_k))$-realization of $(f_1, \dots, f_k)$.*

Proof: By assumption for $i = 1, \dots, k$,

$$x \in \beta(u) \cap \mathrm{dom}(f_i) \Longrightarrow r_i(u) \neq \emptyset \text{ and } \forall v \in r_i(u). f_i(x) \cap \gamma_i(v) \neq \emptyset .$$

Assume, $x \in \beta(u) \cap \mathrm{dom}(f_1, \dots, f_k)$. Then for all $i = 1, \dots, k$: $x \in \beta(u) \cap \mathrm{dom}(f_i)$, hence $r_i(u) \neq \emptyset$, $\forall v \in r_i(u)$ and $f_i(x) \cap \gamma_i(v) \neq \emptyset$. Therefore, $(r_1, \dots, r_k)(u) \neq \emptyset$ and for all $(v_1, \dots, v_k) \in r_1(u) \times \dots \times r_k(u)$,

$$f_1(x) \times \dots \times f_k(x) \ \cap \ \gamma_1(v_1) \times \dots \times \gamma_k(v_k) \neq \emptyset ,$$

hence

$$(f_1, \dots, f_k)(x) \ \cap \ (\gamma_1 \times \dots \times \gamma_k)(v_1, \dots, v_k) \neq \emptyset .$$

□

As expected, the composition of “multi-realized multi-functions” is realized by the composition of realizations (Figure 4).

$$
\begin{tikzcd}
U \arrow[r, Rightarrow, "r"] \arrow[d, Rightarrow, "\beta"'] & V \arrow[r, Rightarrow, "s"] \arrow[d, Rightarrow, "\gamma"'] & W \arrow[d, Rightarrow, "\delta"'] \\
X \arrow[r, Rightarrow, "g"] & Y \arrow[r, Rightarrow, "h"] & Z
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
U \arrow[r, Rightarrow, "s \circ r"] \arrow[d, Rightarrow, "\beta"'] & W \arrow[d, Rightarrow, "\delta"] \\
X \arrow[r, Rightarrow, "h \circ g"] & Z
\end{tikzcd}
$$

Figure 4: Composition of realizations

Lemma 20 *(realization of composition). Let $\beta : U \rightrightarrows X$, $\gamma : V \rightrightarrows Y$ and $\delta : W \rightrightarrows Z$ be generalized multi-representations. If $r : U \rightrightarrows V$ is a (β,γ)-realization of $g : X \rightrightarrows Y$ and $s : V \rightrightarrows W$ is a (γ,δ)-realization of $h : Y \rightrightarrows Z$, then $s \circ r : U \rightrightarrows W$ is a (β,δ)-realization of $h \circ g : X \rightrightarrows Z$.*

Proof: By assumption,

$$x \in \beta(u) \cap \mathrm{dom}(g) \implies r(u) \neq \emptyset \ \text{ and } \ \forall v \in r(u).g(x) \cap \gamma(v) \neq \emptyset \tag{23}$$

$$y \in \gamma(v) \cap \mathrm{dom}(h) \implies s(v) \neq \emptyset \ \text{ and } \ \forall w \in s(v).h(y) \cap \delta(w) \neq \emptyset\,. \tag{24}$$

Assume

$$x \in \beta(u) \cap \mathrm{dom}(h \circ g)\,. \tag{25}$$

We have to show

$$s \circ r(u) \neq \emptyset \ \text{ and} \tag{26}$$

$$\forall w \in s \circ r(u).\, \delta(w) \cap h \circ g(x) \neq \emptyset\,. \tag{27}$$

By (25),

$$x \in \mathrm{dom}(g) \ \text{ and } \ g(x) \subseteq \mathrm{dom}(h)\,. \tag{28}$$

By (23,25,28), $r(u) \neq \emptyset$ and $\forall v \in r(u).g(x) \cap \gamma(v) \neq \emptyset$, hence by (28),

$$r(u) \neq \emptyset \ \text{ and } \ \forall v \in r(u).\exists y \in g(x).y \in \mathrm{dom}(h) \cap \gamma(v)\,. \tag{29}$$

By (24,29) $\forall v \in r(u).s(v) \neq \emptyset$. This proves (26). Also by (24,29),

$$\forall v \in r(u).\exists y \in g(x).\forall w \in s(v).h(y) \cap \delta(w) \neq \emptyset\,. \tag{30}$$

For showing (27) assume $\overline{w} \in s \circ r(u)$. Then $\overline{v} \in r(u)$ and $\overline{w} \in s(\overline{v})$ for some $\overline{v} \in V$. By (30), $\exists y \in g(x).h(y) \cap \delta(\overline{w}) \neq \emptyset$, hence $h \circ g(x) \cap \delta(\overline{w}) \neq \emptyset$. This proves (27). □

Corollary 21 *(multi-variate composition). Let $\beta_i : U_i \rightrightarrows X_i$ $(i = 1, \ldots, k)$, $\gamma_j : V_j \rightrightarrows Y_j$ $(j = 1, \ldots, l)$ and $\delta : W \rightrightarrows Z$ be generalized multi-representations.*

1. *Let* $r_j : U_1 \times \ldots \times U_k \rightrightarrows V_j$ *be a* $((\beta_1 \times \ldots \times \beta_k), \gamma_j)$*-realization of* $g_j : X_1 \times \ldots \times X_k \rightrightarrows Y_j$ $(j = 1, \ldots, l)$ *and*
2. *let* $s : V_1 \times \ldots \times V_l \rightrightarrows W$ *be a* $((\gamma_1 \times \ldots \times \gamma_l), \delta)$*-realization of* $f : Y_1 \times \ldots \times Y_l \rightrightarrows Z$.

Then $s \circ (r_1, \ldots, r_l)$ *is a* $((\beta_1 \times \ldots \times \beta_k), \delta)$*-realization of* $f \circ (g_1, \ldots, g_l)$.

Proof: By Lemma 19 $(r_1, \ldots, r_l)$ is a $((\beta_1 \times \ldots \times \beta_k), (\gamma_1 \times \ldots \times \gamma_l))$-realization of $(g_1, \ldots, g_l)$. The statement follows from Lemma 20 $\square$

We will prove a much more general theorem in Section 7. Realization is downwards transitive. If h realizes g and g realizes f then h realizes f w.r.t. the composed representations (Figure 5).

Lemma 22. *Let* $\gamma : X \rightrightarrows Y$, $\delta : Y \rightrightarrows Z$, $\gamma' : U \rightrightarrows V$ *and* $\delta' : V \rightrightarrows W$ *be generalized multi-representations. If* $h : X \rightrightarrows U$ *is a* (γ, γ')*-realization of* $g : Y \rightrightarrows V$ *and* $g : Y \rightrightarrows V$ *is a* (δ, δ')*-realization of* $f : Z \rightrightarrows W$, *then* $h : X \rightrightarrows U$ *is a* $(\delta \odot \gamma, \delta' \odot \gamma')$*-realization of* $f : Z \rightrightarrows W$.

$$
\begin{tikzcd}
X \arrow[r, Rightarrow, "h"] \arrow[d, Rightarrow, "\gamma"'] & U \arrow[d, Rightarrow, "\gamma'"] \\
Y \arrow[r, Rightarrow, "g"] \arrow[d, Rightarrow, "\delta"'] & V \arrow[d, Rightarrow, "\delta'"] \\
Z \arrow[r, Rightarrow, "f"] & W
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[r, Rightarrow, "h"] \arrow[d, Rightarrow, "\delta \odot \gamma"'] & U \arrow[d, Rightarrow, "\delta' \odot \gamma'"] \\
Z \arrow[r, Rightarrow, "f"] & W
\end{tikzcd}
$$

Figure 5: Realization is transitive downwards.

Proof: Consider $x \in X$ and $z \in Z$ such that

$$z \in \delta \odot \gamma(x) \wedge z \in \mathrm{dom}(f)\,. \tag{31}$$

We have to show

$$x \in \mathrm{dom}(h) \wedge \forall u \in h(x).f(z) \cap \delta' \odot \gamma'(u) \neq \emptyset\,. \tag{32}$$

By assumption for all $y \in Y$,

$$y \in \gamma(x) \wedge y \in \mathrm{dom}(g) \Longrightarrow x \in \mathrm{dom}(h) \wedge \forall u \in h(x).g(y) \cap \gamma'(u) \neq \emptyset \tag{33}$$

$$z \in \delta(y) \wedge z \in \mathrm{dom}(f) \Longrightarrow y \in \mathrm{dom}(g) \wedge \forall v \in g(y).f(z) \cap \delta'(v) \neq \emptyset\,. \tag{34}$$

By (31) for some $y \in Y$,

$$z \in \delta(y) \wedge y \in \gamma(x)\,. \tag{35}$$

By (34,31,35),

$$y \in \mathrm{dom}(g) \wedge \forall v \in g(y).f(z) \cap \delta'(v) \neq \emptyset\,. \tag{36}$$

By (33,35,36),

$$x \in \mathrm{dom}(h) \wedge \forall u \in h(x).g(y) \cap \gamma'(u) \neq \emptyset\,. \tag{37}$$

Thus the first part of (32) has been proved. For showing the second part suppose $u \in h(x)$. By (37) there is some $v \in V$ such that

$$v \in g(y) \cap \gamma'(u)\,. \tag{38}$$

By (36), $f(z) \cap \delta'(v) \neq \emptyset$ and by (38), $f(z) \cap \delta' \odot \gamma'(u) \neq \emptyset$. Thus also the second part of (32) has been proved. □

Notice that for multi-representations we apply the relational composition $\odot$, while for realizing and realized multi-functions in Lemma 20 we apply the stricter "multi-function composition" $\circ$. According to the different interpretations of the concept of multi-function we use different kinds of composition.

7 Flowcharts Realizing Flowcharts

In this section we generalize Lemma 20 from composition to flowcharts with indirect addressing on represented sets. We also generalize [Wei05, Theorem 8.1] from multi-representations to generalized multi-representations.

Let F be a flowchart scheme as given in Definition 9. Remember that $v_1, \ldots, v_m$ are the input variables and v_0 is the output variable of F. Let γ^{var} : **Pointer** $\rightarrow$ **Pointer** and γ^{bool} : **Bool** $\rightarrow$ **Bool** be the identity on Var and $\{0, 1\}$, respectively. Let GR be a finite set of generalized multi-representations $\gamma : \tau \rightrightarrows \tau'$ on types such that $\gamma^{var}, \gamma^{bool} \in \mathrm{GR}$, no other element of GR operates on **Pointer** or **Bool**, that is,

$$\tau, \tau' \notin \{\mathbf{Pointer}, \mathbf{Bool}\} \text{ if } \gamma : \tau \rightrightarrows \tau' \text{ and } \gamma \notin \{\gamma^{var}, \gamma^{bool}\}\,, \text{ and} \tag{39}$$

$$\gamma = \gamma_1 \text{ if } \gamma : \tau \rightrightarrows \tau' \text{ and } \gamma_1 : \tau_1 \rightrightarrows \tau'\,. \tag{40}$$

Theorem 23. *Let (F, I) and (F, J) be flowcharts such that*

1. *for every $v \in \mathrm{Var}$ there is some $\gamma_v : \tau_v \rightrightarrows \tau'_v$ in GR, such that $J(v) = \tau_v$ and $I(v) = \tau'_v$,*

2. for every function symbol f in F there are functions $\delta_i : \tau_i \rightrightarrows \tau'_i$ $(i = 0, \dots, \mu(f))$ in GR *such that*

$$f^J : \tau_1 \times \dots \times \tau_{\mu(f)} \rightrightarrows \tau_0,$$
$$f^I : \tau'_1 \times \dots \times \tau'_{\mu(f)} \rightrightarrows \tau'_0$$

and f^J is a $(\delta_1 \times \dots \times \delta_{\mu(f)}, \delta_0)$-realization of f^I.

Then $f^J_F : \tau_{v_1} \times \dots \times \tau_{v_m} \rightrightarrows \tau_{v_0}$ is a $(\gamma_{v_1} \times \dots \times \gamma_{v_m}, \gamma_{v_0})$-realization of $f^I_F : \tau'_{v_1} \times \dots \times \tau'_{v_m} \rightrightarrows \tau'_{v_0}$ (see Definitions 9, 10, 11).

Therefore, the function f^I_F computed by the flowchart (F, I) is realized by the function f^J_F computed by the flowchart scheme F interpreted with realizing functions. Figure 6 shows interpretations I and J with states σ' and σ, respectively. The example has the special property "$\sigma'(v) \in \gamma_v \circ \sigma(v)$ if $\sigma'(v)$ exists", which will be the invariant in the proof of Proposition 37. By Definition 10(3), $\gamma_1, \dots, \gamma_{\mu(f)} \notin \{\gamma^{var}, \gamma^{bool}\}$ in Theorem 23.2 above.

Proof: See Appendix B. □

Variable v	$\bar{v}$	v'	v_1	u_0	$\bar{u}$	u_2	w	...
$J(v) = (U_v, \dots)$	$\boldsymbol{N}$	**R**	$\boldsymbol{\Sigma^\omega}$	**Pointer**	**Bool**	**Bool**	**Pointer**	...
$\sigma(v)$	237	-4.3	$\uparrow$	v'	1	0	u_0	...
$I(v) = (X_v, \dots)$	$\boldsymbol{\Sigma^*}$	$\mathbf{R}_<$	**R**	**Pointer**	**Bool**	**Bool**	**Pointer**	...
$\sigma'(v)$	`00a10`	4.3	$\uparrow$	v'	$\uparrow$	0	u_0	...

Figure 6: Interpretations I and J with states

8 Computability Induced by Multi-Representations

In this section we draw our main conclusions from the technical results proved in Sections 5 and 7. By multi-representations, where the "names" are finite or infinite strings, computability and continuity can be transferred from Σ^* and Σ^ω to the represented sets [Wei93, Wei00, Sch03, Wei05].

Definition 24 *(induced effectivity).* 1. For multi-representations $\gamma_i : U_i \rightrightarrows X_i$, $U_i \in \{\Sigma^*, \Sigma^\omega\}$ $(i = 0, \dots, k)$, a multi-function $f : X_1 \times \dots \times X_k \rightrightarrows X_0$ is $(\gamma_1, \dots, \gamma_k, \gamma_0)$-continuous or $(\gamma_1, \dots, \gamma_k, \gamma_0)$-computable, iff it has a continuous or computable $(\gamma_1, \dots, \gamma_k, \gamma_0)$-realization, respectively.

2. For multi-representations $\gamma : U \rightrightarrows X$ and $\gamma' : U' \rightrightarrows X'$ ($U, U' \in \{\Sigma^*, \Sigma^\omega\}$) a function $h :\subseteq U \to U'$ translates (or reduces) γ to γ', iff $\gamma(u) \subseteq \gamma' \circ h(u)$ for all $u \in \mathrm{dom}(\gamma)$. Continuous and computable reducibility and equivalence, respectively, are defined by

$$\begin{aligned}
\gamma \leq_t \gamma' &\iff \text{some continuous } h \text{ translates } \gamma \text{ to } \gamma' \quad (\text{“t-reducible”}),\\
\gamma \leq \gamma' &\iff \text{some computable } h \text{ translates } \gamma \text{ to } \gamma' \quad (\text{“reducible”}),\\
\gamma \equiv_t \gamma' &\iff \gamma \leq_t \gamma' \text{ and } \gamma' \leq_t \gamma \quad (\text{“t-equivalent”}),\\
\gamma \equiv \gamma' &\iff \gamma \leq \gamma' \text{ and } \gamma' \leq \gamma \quad (\text{“equivalent”}).
\end{aligned}$$

3. A point $x \in X_0$ is γ_0-computable, iff $x \in \gamma_0(p)$ for some computable $p \in U_0$.

Lemma 25. *1. A function $h :\subseteq Y \to Y'$ translates γ to γ', iff $X \subseteq X'$ and h is a (γ, γ')-realization of the embedding $\mathbb{1}_{XX'}$.*

2. If $f : X \rightrightarrows X'$ is (γ, γ')-computable and $x \in \mathrm{dom}(f)$ is γ-computable, then $y \in f(x)$ for some γ'-computable point $y \in X'$.

The proof follows immediately from the definitions. The multi-functions computable relative to multi-representations are closed under composition.

Corollary 26. *If g_j is $(\gamma_1, \ldots, \gamma_k, \delta_j)$-continuous (-computable) for $j = 1, \ldots, n$ and f is $(\delta_1, \ldots, \delta_n, \delta_0)$-continuous (-computable), then $f \circ (g_1, \ldots, g_n)$ is $(\gamma_1, \ldots, \gamma_k, \delta_0)$-continuous (-computable).*

Proof: This follows from Corollary 21 and the fact that the continuous functions as well as the computable functions on Σ^* and Σ^ω are closed under composition (Lemma 4). □

By Definition 24, every multi-representation of a set X induces a computability concept for functions from and to$X$ Equivalent multi-representations of X induce the same computability on X.

Corollary 27. *Two multi-representations of a set X induce the same kind of continuity (computability) on X, iff they are t-equivalent (equivalent).*

Proof: The “if” follows from Lemma 25 and Corollary 26. Let $\gamma : U \rightrightarrows X$ and $\gamma' : U' \rightrightarrows X$ be multi-representations such that $\gamma \not\leq \gamma'$. Then $\mathbb{1}_{XX}$ is (γ, γ)-computable but not (γ, γ')-computable by Lemma 25. Therefore, γ and γ' induce different kinds of computability on the set X. Continuity is treated correspondingly. □

The next lemma summarizes some further useful observations. The proofs are straightforward.

Lemma 28. *1. For multi-representations γ, δ, if $\gamma \leq_t \delta$ and δ is single-valued, then γ is single-valued.*

2. Let h be a (γ, δ)-realization of a multi-function f. If $\mathrm{card}(\gamma^{-1}[\{x\}]) = 1$ for all $x \in \mathrm{dom}(f)$ and δ is single-valued, then h realizes a single-valued extension of f.

3. Every notation $\nu : \Sigma^* \rightrightarrows M$ has an equivalent representation $\delta_\nu : \Sigma^\omega \rightrightarrows M$. (Define $\delta_\nu(\iota(w)0^\omega) := \nu(w)$.)

By (1) any t-equivalence class contains only single-valued representations or only properly multi-valued representations. By (3) we may replace notations ν by their equivalent representation whenever convenient. For example, a realizing flowchart operating on names from Σ^* and Σ^ω can be replaced by a realizing flowchart operating only on Σ^ω to which Lemma 14 can be applied.

Definition 29. Call a multi-function on *types* (X_i, γ_i) that are multi-represented sets computable, if it has a computable realization and continuous, if it has a continuous realization with respect to these representations.

Notice that this realization is a function on Σ^* or Σ^ω. From Theorem 23 we obtain as a main result that the computable functions on represented sets are closed under flowchart programming.

Theorem 30. *Let (F, I) be a flowchart on multi-represented sets such that each occurring multi-function is computable. Then the multi-function f_F^I computed by the flowchart (F, I) is computable.*

Proof: By Lemma 28(3) we may assume that the sets are represented by Σ^ω. According to Definition 10 let I be an interpretation of F with the types **Pointer** or **Bool** and types τ', which are multi-represented sets (X, δ), $\delta : \Sigma^\omega \rightrightarrows X$. For applying Theorem 23 for every variable v let $\gamma_v := \gamma^{bool}$ if $\tau' = \textbf{Bool}$, $\gamma_v := \gamma^{var}$ if $\tau' = \textbf{Pointer}$ and $\gamma_v : \boldsymbol{\Sigma^\omega} \rightrightarrows \tau'$ with $\gamma(p) = \delta(p)$ if $\tau' = (X, \delta)$. Since each function used for the interpretation I is computable, it has a computable realization on Σ^ω and Σ^*. These realizing functions can be used for an interpretation J of the flowchart scheme F such that the conditions of Theorem 23 hold true. Then by Theorem 23 and Theorem 15, the function f_F^I is computable (on represented sets, Definition 29). $\square$

In many applications Theorem 30 cannot be applied but algorithms on names of points must be written. Very often, however, it is not necessary to return to programming on Σ^* and Σ^ω. The next theorem shows how represented sets with computable functions can be used instead. Consider $X = U = \Sigma^\omega$ in Lemma 22 and Figure 5. The function f is $(\delta \odot \gamma, \delta' \odot \gamma')$-computable, iff f

can be realized (w.r.t. the generalized multi-representations (δ, δ')) by a (γ, γ')-computable function g. Therefore, if we understand computability on (Y, γ), $\gamma : \Sigma^\omega \rightrightarrows Y$, sufficiently well, we can prove computability on $(Z, \delta \odot \gamma)$ by finding computable functions on Y realizing functions on Z w.r.t. the generalized multi-representation $\delta : Y \rightrightarrows Z$. Theorem 31 extends this idea to flowchart computations.

Theorem 31. *Let (F, I) and (F, J) be flowcharts as in Theorem 23. Suppose that in addition for every variable v, the type τ_v is* **Pointer** *or* **Bool** *or a multi-represented set (T_v, β_v), $\beta_v : \Sigma^\omega \rightrightarrows T_v$. If all the multi-functions f_F^J are computable then f_F^I is $(\gamma_{v_1} \odot \beta_{v_1}, \ldots, \gamma_{v_m} \odot \beta_{v_m}, \gamma_{v_0} \odot \beta_{v_0})$-computable.*

Proof: Immediately from Theorems 23, 30 and Lemma 22 □

Theorems 30 and 31 are very useful tools for proving computability of multi-functions on multi-represented sets. The user is no longer restricted to express everything in "0s and 1s" and to describe how Turing machines (or Type-2 machines) operate on codes. By Theorem 30 multi-functions already known to be computable can be combined in flowcharts to compute new multi-functions. Every flowchart containing only computable functions defines a computable function. Thus the user may think in terms of computable operations on real numbers, open sets, continuous functions etc. without using details of the definitions of the representations.

If in some applications details of the representations have to be considered, Theorem 31 allows to do this on some abstract level. As an example consider the Cauchy representation $\rho :\subseteq \Sigma^\omega \to \mathbb{R}$ of the real numbers. It can be decomposed into a representation $\gamma :\subseteq \Sigma^\omega \to \mathbb{Q}^\omega$ of the sequences of rational numbers and a generalized representation $\lambda :\subseteq \mathbb{Q}^\omega \to \mathbb{R}$ computing the limit of fast converging Cauchy sequences, $\rho = \lambda \odot \gamma$. If the user understands computability on $\mathbb{Q}^\omega$ sufficiently well he can avoid to think and speak about codes of rational numbers using 0s and 1s. Both theorems justify and generalize methods which are already used informally in computable analysis.

9 Exponentiation

There are some canonical operators for constructing new representations from given ones such as restriction, conjunction, Cartesian product and exponentiation. Such constructions are used in higher level programming languages for defining new data types. Exponentiation is the most interesting one. For single-valued functions on represented or on multi-represented sets it has been studied to some extent in [Wei00, Sch02, Sch03].

In the following we generalize the type conversion theorem for exponentiation [Wei00, Theorem 3.3.15], [Sch02, Sch03] to multi-functions on multi-represented sets. As we have already mentioned in the discussion of Definition 17, for multi-functions our concept of realization differs from that one in [Sch03].

First, we generalize the definition of $[\gamma_1 \to \gamma_2]$, the standard representation of the (total) (γ_1, γ_2)-continuous functions for single-valued representations γ_1 and γ_2 [KW85][Wei00, Definition 3.3.13][Sch02, Sch03] to multi-representations and multi-functions starting from Definition 17 of realization. For convenience we consider only multi-representations with infinite names (Lemma 28(3)).

Let $\eta : \Sigma^\omega \to \mathbb{F}$ be the standard representation of the set $\mathbb{F}$ of partial continuous functions $h :\subseteq \Sigma^\omega \to \Sigma^\omega$ with G_δ-domain (a set is G_δ if it is a countable intersection of open sets). It is an analogue to an "admissible Gödel numbering" φ of the partial recursive functions [Rog67]. In [Wei00] η is defined as follows: $\eta_{\langle w,q\rangle}(p)$ is the result of the Type-2 machine with code w on input $(p, q) \in \Sigma^\omega \times \Sigma^\omega$. An equivalent representation η' can be defined as follows: $\eta'_q := T_\omega(h)$ where q is interpreted as a listing of graph(h) of a monotone word function $h :\subseteq \Sigma^* \to \Sigma^*$ (Definition 2). For η we have the utm-theorem (the universal function u_η, $u_\eta(q, p) := \eta_q(p)$, is Turing computable), the computable smn-theorem (for every Turing computable function $f :\subseteq \Sigma^\omega \to \Sigma^\omega$ there is a computable function $s : \Sigma^\omega \to \Sigma^\omega$ such that $f\langle p, q\rangle = \eta_{s(p)}(q)$), and the continuous smn-theorem (for every function $f \in \mathbb{F}$ there is a continuous function $s : \Sigma^\omega \to \Sigma^\omega$ such that $f\langle p, q\rangle = \eta_{s(p)}(q)$) [Wei00].

Definition 32. For multi-representations $\delta_i : \Sigma^\omega \rightrightarrows X_i$ $(i = 1, 2)$ define

1. the product representation $[\delta_1, \delta_2] : \Sigma^\omega \rightrightarrows X_1 \times X_2$ by
   $[\delta_1, \delta_2]\langle p_1, p_2\rangle := \delta_1(p_1) \times \delta_2(p_2)$,
2. the multi-representation $[\delta_1 \rightrightarrows \delta_2]$ of the set $\mathrm{CM}(\delta_1, \delta_2)$ of the (δ_1, δ_2)-continuous multi-functions $f : X_1 \rightrightarrows X_2$ by
   $[\delta_1 \rightrightarrows \delta_2](p) = f \iff \eta_p$ realizes f w.r.t. (δ_1, δ_2).
3. Let $[\delta_1 \to_p \delta_2](p)$ be the restriction of $[\delta_1 \rightrightarrows \delta_2]$ to the set $\mathrm{CP}(\delta_1, \delta_2)$ of the partial (δ_1, δ_2)-continuous functions $f :\subseteq X_1 \to X_2$.
4. Let $[\delta_1 \to \delta_2](p)$ be the restriction of $[\delta_1 \rightrightarrows \delta_2]$ to the set $\mathrm{C}(\delta_1, \delta_2)$ of the total (δ_1, δ_2)-continuous functions $f : X_1 \to X_2$.

(See Definition 17 and Figure 3). Notice that $[\delta_1 \to \delta_2]$ is multi-valued in general and that even for single-valued representations δ_i the representations $[\delta_1 \rightrightarrows \delta_2]$ and $[\delta_1 \to_p \delta_2]$ are multi-valued in general, since η_p realizes every restriction of f if it realizes f (Lemma 18). We generalize Theorem 3.3.15 in [Wei00] on type conversion as follows.

Theorem 33 *(type conversion). Let $\delta_X : \Sigma^\omega \rightrightarrows X$, $\delta_Y : \Sigma^\omega \rightrightarrows Y$, and $\delta_Z : \Sigma^\omega \rightrightarrows Z$ be multi-representations. Define an operator T from the $([\delta_X, \delta_Y], \delta_Z)$-continuous multi-functions $f : X \times Y \rightrightarrows Z$ to the set of total functions from X to the multi-functions from Y to Z by*

$$T(f)(x)(y) := f(x, y). \tag{41}$$

For $\gamma_1 := [[\delta_X, \delta_Y] \rightrightarrows \delta_Z]$ and $\gamma_2 := [\delta_X \to [\delta_Y \rightrightarrows \delta_Z]]$,

1. *T is (γ_1, γ_2)-computable and T^{-1} is (γ_2, γ_1)-computable.*
2. $T \circ \gamma_1 \equiv \gamma_2$,
3. *A function f is $([\delta_X, \delta_Y], \delta_Z)$-computable, iff $T(f)$ is $(\delta_X, [\delta_Y \rightrightarrows \delta_Z])$-computable.*

Notice that $T(f)$ is single-valued for each $f : X \times Y \rightrightarrows Z$. In programming, the operator T can be expressed by λ abstraction, by (41) formally,

$$T(f) = \lambda x.\lambda y.f(x, y) .$$

The theorem generalizes the utm-theorem and the smn-theorem for η. The occurrence of *total* functions from X in γ_2 corresponds to the totality of the index function s in the smn-theorem for η.

Proof: For any $f : X \times Y \rightrightarrows Z$ and $p, s \in \Sigma^\omega$,

$$\begin{aligned}
& f \in \gamma_1(p) \\
\iff & f \in [[\delta_X, \delta_Y] \rightrightarrows \delta_Z](p) \\
\iff & (\forall x, y)(\forall q, r)\big((x, y) \in \mathrm{dom}(f) \cap [\delta_X, \delta_Y]\langle q, r\rangle \\
& \quad \Longrightarrow f(x, y) \cap \delta_Z \eta_p \langle q, r\rangle \neq \emptyset\big).
\end{aligned} \tag{42}$$

and

$$\begin{aligned}
& T(f) \in \gamma_2(s) \\
\iff & T(f) \in [\delta_X \to [\delta_Y \rightrightarrows \delta_Z]](s) \\
\iff & (\forall x, q)\big(x \in \delta_X(q) \Longrightarrow T(f)(x) \in [\delta_Y \rightrightarrows \delta_Z]\eta_s(q)\big) \\
\iff & (\forall x, q)\big(x \in \delta_X(q) \Longrightarrow \\
& \quad (\forall y, r)(y \in \mathrm{dom}(T(f)) \cap \delta_Y(r) \Longrightarrow T(f)(x)(y) \cap \delta_Z \eta_{\eta_s(q)}(r) \neq \emptyset)\big) \\
\iff & (\forall x, q)(\forall y, r)\big(x \in \delta_X(q) \wedge y \in \mathrm{dom}(T(f)(x)) \cap \delta_Y(r) \\
& \quad \Longrightarrow f(x, y) \cap \delta_Z \eta_{\eta_s(q)}(r) \neq \emptyset)\big) \\
\iff & (\forall x, y)(\forall q, r)\big((x, y) \in \mathrm{dom}(f) \cap [\delta_X, \delta_Y]\langle q, r\rangle \\
& \quad \Longrightarrow f(x, y) \cap \delta_Z \eta_{\eta_s(q)}(r) \neq \emptyset)\big).
\end{aligned} \tag{43}$$

By the utm-theorem and the computable smn-theorem for η there are computable functions $a, b : \Sigma^\omega \to \Sigma^\omega$ such that

$$\eta_p\langle q, r\rangle = \eta_{\eta_{a(p)}(q)}(r) \quad \text{and} \quad \eta_{\eta_s(q)}(r) = \eta_{b(s)}\langle q, r\rangle\,.$$

From (42) and (43) we conclude $f \in \gamma_1(p) \;\Rightarrow\; T(f) \in \gamma_2 a(p)$ and $T(f) \in \gamma_2(s) \;\Rightarrow\; f \in \gamma_1 b(s)$, hence, T is (γ_1, γ_2)-computable and T^{-1} is (γ_2, γ_1)-computable. Theorem 33(2) is equivalent to Theorem 33(1). Theorem 33(3) follows from Theorem 33(1), since T is a single-valued computable operator and, therefore, maps computable elements to computable elements. □

For single-valued representations and functions and a representation γ of (δ_X, δ_Y)-continuous functions, the apply function is $(\gamma, \delta_X, \delta_Y)$-computable, iff $\gamma \le [\delta_X \to \delta_Y]$ [Wei00]. We generalize this result as follows.

Corollary 34. *For multi-representations δ_X, δ_Y and a multi-representation γ of multi-functions $f : X \rightrightarrows Y$, the apply multi-function is $(\gamma, \delta_X, \delta_Y)$-computable, iff $\gamma \le [\delta_X \rightrightarrows \delta_Y]$. In particular, the apply multi-function is $([\delta_X \rightrightarrows \delta_Y], \delta_X, \delta_Y)$-computable.*

Proof: For every function $h \in \text{range}(\gamma)$, every $x \in X$ and every $y \in Y$,

$$y \in T(\text{apply})(h)(x) \iff y \in \text{apply}(h, x) \iff y \in h(x).$$

Therefore, $T(\text{apply})$ is the identity on range(γ). By Theorem 33,

$$\begin{aligned} & \text{apply is } (\gamma, \delta_X, \delta_Y)\text{-computable} \\ \iff & T(\text{apply}) \text{ is } (\gamma, [\delta_X \rightrightarrows \delta_Y])\text{-computable} \\ \iff & \gamma \le [\delta_X \rightrightarrows \delta_Y]\,. \end{aligned}$$

Remember that by Definition 24, the identity on range(γ) is (γ, γ')-computable, iff $\gamma \le \gamma'$. □

For multi-functions there is another kind of type conversion. As an example consider a $(\delta_X, \nu_{\mathbb{N}}, \nu_{\mathbb{Q}})$-computable multi-function $f : X \times \mathbb{N} \rightrightarrows \mathbb{Q}$ such that for all $x \in X$, $|a_i - a_j| \le 2^{-i}$, if $i \le j$, $a_i \in f(x, i)$ and $a_j \in f(x, j)$. Then there is a computable multi-function $Sf : X \rightrightarrows \mathbb{Q}^{\mathbb{N}}$ such that $|g(i) - g(j)| \le 2^{-i}$ for every $g \in Sf(x)$. Applying the computable limit operator Lim for fast converging Cauchy sequences [Wei00] we obtain a computable (single-valued) function $\text{Lim} \circ Sf : X \to \mathbb{R}$.

Theorem 35 *(type conversion (ii)). Let $\delta_X, \delta_Y, \delta_Z$ be multi-representations of X, Y and Z, respectively, δ_Y, δ_Z single-valued and δ_Y injective. For the set G_1*

of the $([\delta_X, \delta_Y], \delta_Z)$*-continuous multi-functions and the set* G_2 *of the* $(\delta_X, [\delta_Y \to \delta_Z])$*-continuous multi-functions define* $S : G_1 \to G_2$ *as follows:*

$$\begin{aligned} x \in \mathrm{dom}(Sf) &\iff (\forall\, y \in Y)\; f(x,y) \neq \emptyset \\ g \in Sf(x) &\iff (\forall\, y \in Y)\; g(y) \in f(x,y)\,. \end{aligned}$$

Then for $\gamma_1 := [[\delta_X, \delta_Y] \rightrightarrows \delta_Z]$ *and* $\gamma_2 := [\delta_X \rightrightarrows [\delta_Y \to \delta_Z]]$*,* S *is* (γ_1, γ_2)*-computable.*

The operator S is single-valued but not injective in general since $S(f) = S(f|_{\{x \mid (\forall y) f(x,y) \neq \emptyset\}})$.

Proof: By the utm-theorem and the smn-theorem for η there is a computable function $a : \Sigma^\omega \to \Sigma^\omega$ such that

$$\eta_p\langle q, r\rangle = \eta_{\eta_{a(p)}(q)}(r)\,.$$

For $f \in G_1$,

$$\begin{aligned} f \in \gamma_1(p) \Longrightarrow\; & (\forall x, y, q, r)[(x,y) \in \mathrm{dom}(f) \wedge x \in \delta_X(q) \wedge \delta_Y(r) = y \\ & \qquad\qquad \Longrightarrow \delta_Z \eta_p\langle q, r\rangle \in f(x,y)] \\ \Longrightarrow\; & (\forall x, q)[x \in \mathrm{dom}(Sf) \wedge x \in \delta_X(q) \\ & \qquad\qquad \Longrightarrow (\forall y, r)[\delta_Y(r) = y \Longrightarrow \delta_Z \eta_{\eta_{a(p)}(q)}(r) \in f(x,y)]] \end{aligned}$$

If we abbreviate $h(y) := \delta_Z \eta_{\eta_{a(p)}(q)}(\delta_Y^{-1}(y))$, then $h\delta_Y(r) = \delta_Z \eta_{\eta_{a(p)}(q)}(r)$ and hence $h = [\delta_Y \to \delta_Z]\eta_{a(p)}(q)$. We obtain

$$\begin{aligned} f \in \gamma_1(p) \Longrightarrow\; & (\forall x, q)[x \in \mathrm{dom}(Sf) \wedge x \in \delta_X(q) \\ & \quad \Longrightarrow (\forall y, r)[\delta_Y(r) = y \Longrightarrow ([\delta_Y \to \delta_Z]\eta_{a(p)}(q)(y)) \in f(x,y)]] \\ \Longrightarrow\; & (\forall x, q)[x \in \mathrm{dom}(Sf) \wedge x \in \delta_X(q) \\ & \quad \Longrightarrow (\forall y)[([\delta_Y \to \delta_Z]\eta_{a(p)}(q)(y)) \in f(x,y)]] \\ \Longrightarrow\; & (\forall x, q)[x \in \mathrm{dom}(Sf) \wedge x \in \delta_X(q) \\ & \quad \Longrightarrow [\delta_Y \to \delta_Z]\eta_{a(p)}(q) \in Sf(x)] \\ \Longrightarrow\; & Sf \in [\delta_X \rightrightarrows [\delta_Y \to \delta_Z]]a(p) \in \gamma_2 a(p)\,. \end{aligned}$$

Therefore, S is (γ_1, γ_2)-computable. □

10 Final Remarks

For simplicity the results in this article have been formulated and proved for flowcharts with indirect addressing. By the remarks at the end of Section 4 thy apply to flowcharts with direct and indirect addressing as well.

Most representations, which are of interest to some "users", are very elementary or can be constructed from simple ones by operations such as "restriction", "factorization", "Cartesian product", "set of finite subsets", "countable conjunction" or "λ-abstraction", see Definition 32, Section 3.3 in [Wei00], [Sch02]. By Corollary 27 two multi-representations induce the same computability, iff they are equivalent. Therefore, the computability concept on a set X is characterized by an equivalence class E of multi-representations of X. Let a *c-type* (computation type) be a pair $\tau = (X, E)$ and call a function $f : \tau_1 \rightrightarrows \tau_2$ computable, iff it is (δ_1, δ_2)-computable for some $\delta_1 \in E_1$ and $\delta_2 \in E_2$. The operations on multi-representations mentioned above map equivalent representations to equivalent ones, hence can be defined on c-types.

It is known that important elementary c-types can be characterized without mentioning representations by requiring that some functions become computable, and that some of the constructions on c-types can be characterized by requiring that some functions from, on or to the new c-types are computable [Bra99, Her99]. This shows the possibility to define important c-types abstractly and to write programs operating on them without mentioning representations at all. For such programming systems it should be possible to prove not only soundness but completeness as well. However, the final implementation of such a program on a computer (Turing machine) must map sequences of symbols and therefore must be specialized to single representatives of the equivalence classes of representations. Furthermore, the user must "understand" at least the input and output representations. Otherwise he cannot feed the machine with meaningful data and cannot understand its results.

Acknowledgement
The author wants to thank the unknown referees for valuable comments and suggestions.

References

[BC90] Boehm, H, Cartwright, R.: Exact real arithmetic, formulating real numbers as functions. In D. Turner, editor, *Research topics in functional programming*, pages 43–64. Addison-Wesley, 1990.

[BCSS98] Blum L., Cucker, F., Shub, M., Smale, S.: *Complexity and Real Computation*. Springer, New York, 1998.

[BH98] Brattka, V., Hertling, P.: Feasible real random access machines. *Journal of Complexity*, 14(4):490–526, 1998.

[Bla97] Blanck, J.: Domain representability of metric spaces. *Annals of Pure and Applied Logic*, 83:225–247, 1997.

[Bla00] Blanck, J.: Domain representations of topological spaces. *Theoretical Computer Science*, 247:229–255, 2000.

[Bra96] Brattka, V.: Recursive characterization of computable real-valued functions and relations. *Theoretical Computer Science*, 162:45–77, 1996.

[Bra99] Brattka, V.: Computable invariance. *Theoretical Computer Science*, 210:3–20, 1999.

[Bra03] Brattka, V.: Computability over topological structures. In S. Barry Cooper and Sergey S. Goncharov, editors, *Computability and Models*, pages 93–136. Kluwer Academic Publishers, New York, 2003.

[BSHT02] Blanck, J., Stoltenberg-Hansen, V., Tucker, J.V.: Domain representations of partial functions, with applications to spatial objects and constructive volume geometry. *Theoretical Computer Science*, 284(2):207–240, 2002.

[BSS89] Blum, L., Shub, M., Smale, S.: On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society*, 21(1):1–46, July 1989.

[CDG06] Ciaffaglione, A., Di Gianantonio, P.: A certified, corecursive implementation of exact real numbers. *Theoretical Computer Science*, 351(1):39–51, 2006.

[Col05] Collins, P.: Continuity and computability on reachable sets. *Theoretical Computer Science*, 341:162–195, 2005.

[DG99] Di Gianantonio, P.: An abstract data type for real numbers. *Theoretical Computer Science*, 221:295–326, 1999.

[EE01] Edalat, A., Hötzel-Escardó, M..: Integration in real PCF. *Information and Computation*, 160:128–166, 2001.

[EMR07] Hötzel-Escardó, M., Marcial-Romero, J.R.: Semantics of a sequential language for exact real-number computation. *Theoretical Computer Science*, 379:120–141, 2007.

[ES99] Hötzel-Escardó, M., Streicher, T.: Induction and recursion on the partial real line with applications to real PCF. *Theoretical Computer Science*, 210:121–157, 1999.

[Esc96] Hötzel-Escardó, M.: PCF extended with real numbers. *Theoretical Computer Science*, 162:79–115, 1996.

[Grz55] Grzegorczyk, A.: Computable functionals. *Fundamenta Mathematicae*, 42:168–202, 1955.

[Grz57] Grzegorczyk, A.: On the definitions of computable real continuous functions. *Fundamenta Mathematicae*, 44:61–71, 1957.

[GW05] Grubba, T., Weihrauch, K.: A computable version of Dini's theorem for topological spaces. In Pinar Yolum, Tunga Güngör, Fikret Gürgen, and Can Özturan, editors, *Computer and Information Sciences - ISCIS 2005*, volume 3733 of *Lecture Notes in Computer Science*, pages 927–936, Berlin, 2005. Springer. 20th International Symposium, ISCIS, Istanbul, Turkey, October 2005.

[Her99] Hertling, P.: A real number structure that is effectively categorical. *Mathematical Logic Quarterly*, 45(2):147–182, 1999.

[HU79] Hopcroft, J.E., Ullman, J.D.:. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, 1979.

[KT84] Klein, E. ,Thompson, A.C.: *Theory of correspondences*. John Wiley & Sons, New York, 1984.

[KW85] Kreitz, C., Weihrauch, K.: Theory of representations. *Theoretical Computer Science*, 38:35–53, 1985.

[Lac55] Lacombe, D.: Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles I-III. *Comptes Rendus Académie des Sciences Paris*, 240,241:2478–2480,13–14,151–153, 1955. Théorie des fonctions.

[Luc77] Luckhardt, H.: A fundamental effect in computations on real numbers. *Theoretical Computer Science*, 5(3):321–324, 1977.

[Rog67] Rogers, H.: *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

[Sch02] Schröder, M.: Extended admissibility. *Theoretical Computer Science*, 284(2):519–538, 2002.

[Sch03] Schröder, M.: Admissible representations for continuous computations. Informatik Berichte 299, FernUniversität Hagen, Hagen, April 2003. Dissertation.

[Sco70] Scott, D.: Outline of a mathematical theory of computation. Technical Monograph PRG-2, Oxford University, Oxford, November 1970.

[SHT99] Stoltenberg-Hansen, V., Tucker, J.V.: Concrete models of computation for topological algebras. *Theoretical Computer Science*, 219:347–378, 1999.

[Ste99] Stewart, K.J.: *Concrete and Abstract Models of Computation over Metric Algebras*. PhD thesis, University of Wales Swansea, Singleton Park, Swansea, SA2 8PP, UK, 1999.

[Tur36] Turing, A.M.: On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[TZ99] Tucker, J.V., Zucker, J.I.: Computation by 'While' programs on topological partial algebras. *Theoretical Computer Science*, 219:379–420, 1999.

[TZ00] Tucker, J.V., Zucker, J.I.: Computable functions and semicomputable sets on many-sorted algebras. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5*, pages 317–523, Oxford, 2000. Oxford University Press.

[TZ04] Tucker, J.V., Zucker, J.I.: Abstract versus concrete computation on metric partial algebras. *ACM Transactions on Computational Logic*, 5(4):611–668, 2004.

[Wei87] Weihrauch, K.: *Computability*, volume 9 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1987.

[Wei93] Weihrauch, K.: Computability on computable metric spaces. *Theoretical Computer Science*, 113:191–210, 1993. Fundamental Study.

[Wei00] Weihrauch, K.: *Computable Analysis*. Springer, Berlin, 2000.

[Wei05] Weihrauch, K.: Multi-functions on multi-represented sets are closed under flowchart programming. In Tanja Grubba, Peter Hertling, Hideki Tsuiki, and Klaus Weihrauch, editors, *Computability and Complexity in Analysis*, volume 326 of *Informatik Berichte*, pages 267–300. FernUniversität in Hagen, July 2005. Proccedings, Second International Conference, CCA 2005, Kyoto, Japan, August 25–29, 2005.

[ZW03] Zhong, N., Weihrauch, K.: Computability theory of generalized functions. *Journal of the Association for Computing Machinery*, 50(4):469–505, 2003.

A Proof of Lemma 14

In this proof we may assume $Q\subseteq\Sigma^*$. Since all the functions f^I and f^J are single-valued, by (8) the next-relations $\stackrel{I}{\vdash}$ and $\stackrel{J}{\vdash}$ are single-valued, i.e. for each input there is at most one maximal computation. For $x \in (\Sigma^*)^m$ and $p \in (\Sigma^\omega)^m$ let x_i and p_i the ith component of x and p, respectively. For $x \in (\Sigma^*)^m$ let $(l_0^x, \sigma_0^x), (l_1^x, \sigma_1^x), \ldots$ be the maximal x-computation on the flowchart (F, J), and for $p \in (\Sigma^\omega)^m$ let $(l_0^{Ip}, \sigma_0^{Ip}), (l_1^{Ip}, \sigma_1^{Ip}), \ldots$ be the maximal p-computation on the flowchart (F, I). In the following proposition we consider the label and the register contents after n steps as functions of the input to the flowchart.

Proposition 36. *For $n \in \mathbb{N}$ and $v \in \mathrm{Var}$,*

1. *The function* $x \mapsto l_n^x$ *is monotone-constant and* $T_*(x \mapsto l_n^x)$ *extends* $p \mapsto l_n^{Ip}$*,*
2. *if* $J(v) \in \{\mathbf{Pointer}, \mathbf{Bool}\}$*, then* $x \mapsto \sigma_n^x(v)$ *is monotone-constant and* $T_*(x \mapsto \sigma_n^x(v))$ *extends* $p \mapsto \sigma_n^{Ip}(v)$*,*
3. *if* $J(v) = \boldsymbol{\Sigma^*}$*, then* $x \mapsto \sigma_n^x(v)$ *is monotone and* $T_\omega(x \mapsto \sigma_n^x(v))$ *extends* $p \mapsto \sigma_n^{Ip}(v)$*,*

Proof: (Proposition 36) (By induction on n)

$\boldsymbol{n = 0}$:
– Since $(\forall z)(x \mapsto l_0^x)(z) = l_{in}$ and $(\forall q)(p \mapsto l_0^{Ip})(q) = l_{in}$, the function $x \mapsto l_0^x$ is monotone-constant and $T_*(x \mapsto l_0^x)$ extends $p \mapsto l_0^{Ip}$.
– For $J(v) = \mathbf{Pointer}$, $(\forall z)(x \mapsto \sigma_0^x(v))(z) = v$ and $(\forall q)(p \mapsto \sigma_0^{Ip}(v))(q) = v$, therefore, $x \mapsto \sigma_0^x(v)$ is monotone-constant and $T_*(x \mapsto \sigma_0^x(v))$ extends $p \mapsto \sigma_0^{Ip}(v)$.
– For $J(v) = \mathbf{Bool}$, $(\forall z)(x \mapsto \sigma_0^x(v))(z) = \uparrow$ and $(\forall q)(p \mapsto \sigma_0^{Ip}(v))(q) = \uparrow$, therefore, $x \mapsto \sigma_0^x(v)$ is monotone-constant and $T_*(x \mapsto \sigma_0^x(v))$ extends $p \mapsto \sigma_0^{Ip}(v)$.
– For $v = v_i$ $(1 \leq i \leq m)$, $(\forall z)(x \mapsto \sigma_0^x(v_i))(z) = z_i$ and $(\forall q)(p \mapsto \sigma_0^{Ip}(v_i))(q) = q_i$, therefore, $x \mapsto \sigma_0^x(v)$ is monotone and $T_\omega(x \mapsto \sigma_0^x(v))$ extends $p \mapsto \sigma_0^{Ip}(v)$
– For $J(v) = \boldsymbol{\Sigma^*}$ and $v \notin \{v_1, \ldots, v_m\}$, $(\forall z)(x \mapsto \sigma_0^x(v))(z) = \uparrow$ and $(\forall q)(p \mapsto \sigma_0^{Ip}(v))(q) = \uparrow$, therefore $x \mapsto \sigma_0^x(v)$ is monotone and $T_\omega(x \mapsto \sigma_0^x(v))$ extends $p \mapsto \sigma_0^{Ip}(v)$.

$\boldsymbol{n \Longrightarrow n+1}$:
Assume that Proposition 36(1) - 36(3) are valid for some n.

(A) monotone-constant/monotone: First, we prove that the functions are monotone-constant or monotone, respectively, for $n+1$. Let $x, y \in (\Sigma^*)^m$ such that $x \sqsubseteq y$. We have to prove

$$l_{n+1}^x = l_{n+1}^y\,, \quad \text{if} \quad l_{n+1}^x \quad \text{exists,} \tag{44}$$

$$\sigma_{n+1}^x(v) = \sigma_{n+1}^y(v) \quad \text{if} \quad J(v) \neq \boldsymbol{\Sigma^*} \quad \text{and} \quad \sigma_{n+1}^x(v) \quad \text{exists,} \tag{45}$$

$$\sigma_{n+1}^x(v) \sqsubseteq \sigma_{n+1}^y(v) \quad \text{if} \quad J(v) = \boldsymbol{\Sigma^*} \quad \text{and} \quad \sigma_{n+1}^x(v) \quad \text{exists.} \tag{46}$$

If l_{n+1}^x exists or $\sigma_{n+1}^x(v)$ exists for some v, then the $(n+1)$st configuration $(l_{n+1}^x, \sigma_{n+1}^x)$ exists. Therefore assume that the configuration $(l_{n+1}^x, \sigma_{n+1}^x)$ exists. Then the configuration (l_n^x, σ_n^x) exists. We investigate the two cases "assignment" and "branching" from Definition 11 separately.

(A1) Assignment:
Suppose, $\mathrm{Stmt}(l_n^x) = (u := f(u_1, \ldots, u_k), \bar{l})$ for some $u, u_1, \ldots, u_k, f, \bar{l}$.

Then by (8),

$$l^x_{n+1} = \bar{l} \text{ and } \sigma^x_{n+1} \circ \sigma^x_n(u) = f^J(\sigma^x_n \circ \sigma^x_n(u_1), \ldots, \sigma^x_n \circ \sigma^x_n(u_k)) . \tag{47}$$

First, we show that $(l^y_{n+1}, \sigma^y_{n+1})$ exists.
By (47), $\sigma^x_n(u') \in \mathrm{Var}$ exists for $u' = u, u_1, \ldots, u_k$, hence by induction hypothesis

$$\sigma^y_n(u') \in \mathrm{Var} \ \text{ exists and } \ \sigma^x_n(u') = \sigma^y_n(u') \ \text{ for } \ u' = u, u_1, \ldots, u_k . \tag{48}$$

Since by (47), $\sigma^x_n \circ \sigma^x_n(u') \in \Sigma^*$ exists for $u' = u_1, \ldots, u_k$, from (48) and by induction hypothesis

$$\sigma^y_n \circ \sigma^y_n(u') \ \text{ exists and } \ \sigma^x_n \circ \sigma^x_n(u') \sqsubseteq \sigma^y_n \circ \sigma^y_n(u') \ \text{ for } \ u' = u_1, \ldots, u_k . \tag{49}$$

Since f^J is (monotone-constant or) monotone, by (47),

$$f^J(\sigma^y_n \circ \sigma^y_n(u_1), \ldots, \sigma^y_n \circ \sigma^y_n(u_k)) \ \text{ exists.} \tag{50}$$

Since by induction hypothesis $l^y_n = l^x_n$, $\mathrm{Stmt}(l^y_n) = (u := f(u_1, \ldots, u_k), \bar{l})$. Therefore by Definition 11(1), $(l^y_{n+1}, \sigma^y_{n+1})$ exists such that

$$l^y_{n+1} = \bar{l} \ \text{ and } \ \sigma^y_{n+1} \circ \sigma^y_n(u) = f^J(\sigma^y_n \circ \sigma^y_n(u_1), \ldots, \sigma^y_n \circ \sigma^y_n(u_k)) . \tag{51}$$

We verify (44 – 46).

From (47) and (51), $l^y_{n+1} = \bar{l} = l^x_{n+1}$. Therefore, (44) is true.

Suppose, $J(v) \neq \boldsymbol{\Sigma^*}$ and $\sigma^x_{n+1}(v)$ exists.

- Suppose, $v \neq \sigma^y_n(u)$ $(= \sigma^x_n(u))$.
  By induction hypothesis, $\sigma^y_n(v) = \sigma^x_n(v)$ and therefore by (9),

$$\sigma^x_{n+1}(v) = \sigma^x_n(v) = \sigma^y_n(v) = \sigma^y_{n+1}(v) .$$

- Suppose, $v = \sigma^y_n(u)$ $(= \sigma^x_n(u))$.
  Since f^J maps to $J \circ \sigma^x_n(u) = J(v) \neq \boldsymbol{\Sigma^*}$, by Lemma 14(2), f^J is monotone-constant. From (47 – 51),

$$\begin{aligned}
\sigma^x_{n+1}(v) &= \sigma^x_{n+1} \circ \sigma^x_n(u) \\
&= f^J(\sigma^x_n \circ \sigma^x_n(u_1), \ldots, \sigma^x_n \circ \sigma^x_n(u_k)) \\
&= f^J(\sigma^y_n \circ \sigma^y_n(u_1), \ldots, \sigma^y_n \circ \sigma^y_n(u_k)) \\
&= \sigma^y_{n+1} \circ \sigma^y_n(u) \\
&= \sigma^y_{n+1}(v) .
\end{aligned}$$

Therefore, (45) is true.

Suppose, $J(v) = \boldsymbol{\Sigma^*}$ and $\sigma^x_{n+1}(v)$ exists.

- Suppose, $v \neq \sigma^y_n(u)$ $(= \sigma^x_n(u))$.
  By induction hypothesis, $\sigma^x_n(v) \sqsubseteq \sigma^y_n(v)$ and therefore by (9),
  $$\sigma^x_{n+1}(v) = \sigma^x_n(v) \sqsubseteq \sigma^y_n(v) = \sigma^y_{n+1}(v)\,.$$
- Suppose, $v = \sigma^y_n(u)$ $(= \sigma^x_n(u))$.
  Since f^J maps to $J \circ \sigma^x_n(u) = J(v) = \boldsymbol{\Sigma^*}$, by Lemma 14(2), f^J is monotone.
  From (47 – 51),
  $$\begin{aligned} \sigma^x_{n+1}(v) &= \sigma^x_{n+1} \circ \sigma^x_n(u) \\ &= f^J(\sigma^x_n \circ \sigma^x_n(u_1), \ldots, \sigma^x_n \circ \sigma^x_n(u_k)) \\ &\sqsubseteq f^J(\sigma^y_n \circ \sigma^y_n(u_1), \ldots, \sigma^y_n \circ \sigma^y_n(u_k)) \\ &= \sigma^y_{n+1} \circ \sigma^y_n(u) \\ &= \sigma^y_{n+1}(v)\,. \end{aligned}$$

Therefore, (46) is true.
Thus we have proved (44-46) for assignments.

(A2) Branching:
Suppose, $\mathrm{Stmt}(l^x_n) = (\text{if } u \text{ then } l', l'')$ for some u, l', l''.
By induction hypothesis, $l^x_n = l^y_n$, therefore, $\mathrm{Stmt}(l^y_n) = (\text{if } u \text{ then } l', l'')$. Since $\sigma^x_n \circ \sigma^x_n(u)$ exists, $J(u) = \textbf{Pointer}$ and $J \circ \sigma^x_n(u) = \textbf{Bool}$. By induction hypothesis, $\sigma^x_n(u) = \sigma^y_n(u)$ and $\sigma^x_n \circ \sigma^x_n(u) = \sigma^y_n \circ \sigma^x_n(u) = \sigma^y_n \circ \sigma^y_n(u)$ We obtain

$$l^x_{n+1} = \left\{ \begin{array}{l} l' \text{ if } \;\sigma^x_n \circ \sigma^x_n(u) = \mathsf{1} \\ l'' \text{ if } \;\sigma^x_n \circ \sigma^x_n(u) = \mathsf{0} \end{array} \right\} = \left\{ \begin{array}{l} l' \text{ if } \;\sigma^y_n \circ \sigma^y_n(u) = \mathsf{1} \\ l'' \text{ if } \;\sigma^y_n \circ \sigma^y_n(u) = \mathsf{0} \end{array} \right\} = l^y_{n+1}\,.$$

Therefore (44) is true.
Suppose, $\sigma^x_{n+1}(v)$ exists. Since $\sigma^x_{n+1} = \sigma^x_n$ and $\sigma^y_{n+1} = \sigma^y_n$, by induction hypothesis

$$\sigma^x_{n+1}(v) = \sigma^x_n(v) \left\{ \begin{array}{l} = \sigma^y_n(v) = \sigma^y_{n+1}(v) \text{ if } \;I(v) \notin \boldsymbol{\Sigma^*}\,, \\ \sqsubseteq \sigma^y_n(v) = \sigma^y_{n+1}(v) \text{ if } \;I(v) \in \boldsymbol{\Sigma^*}\,. \end{array} \right.$$

Thus we have proved (44-46) (with $n+1$ replaced for n) for branchings.

Therefore, the functions in Proposition 36 are monotone-constant or monotone, respectively, for $n+1$. Next, we prove the extension properties in Proposition 36 for $n+1$.

(B) Extension:
Assume that the extension properties from Proposition 36 are true for n. By Definition 2 it remains to show for arbitrary $q \in (\Sigma^\omega)^m$ and $v \in \mathrm{Var}$:

$$\text{If } \ l^{Iq}_{n+1} \ \text{ exists, then } \ (\exists y \sqsubseteq q)\, l^y_{n+1} = l^{Iq}_{n+1}\,. \tag{52}$$

$$\begin{array}{l}\text{If } \ I(v) \in \{\mathbf{Pointer}, \mathbf{Bool}\} \text{ and } \ \sigma^{Iq}_{n+1}(v) \ \text{ exists, then} \\ \qquad (\exists y \sqsubseteq q)\, \sigma^y_{n+1}(v) = \sigma^{Iq}_{n+1}(v)\,.\end{array} \tag{53}$$

$$\begin{array}{l}\text{If } \ I(v) = \boldsymbol{\Sigma^\omega} \ \text{ and } \ \sigma^{Iq}_{n+1}(v) \ \text{ exists, then} \\ \qquad (\forall j)(\exists y \sqsubseteq q)\left(\sigma^y_{n+1}(v) \sqsubseteq \sigma^{Iq}_{n+1}(v) \ \text{ and } \ |\sigma^y_{n+1}(v)| \geq j\right).\end{array} \tag{54}$$

If l^{Iq}_{n+1} or $\sigma^{Iq}_{n+1}(v)$ exists, then the $(n+1)$st configuration $(l^{Iq}_{n+1}, \sigma^{Iq}_{n+1})$ exists. Therefore, assume that $(l^{Iq}_{n+1}, \sigma^{Iq}_{n+1})$ exists. Then $(l^{Iq}_n, \sigma^{Iq}_n)$ exists. We investigate the two cases "assignment" and "branching" from Definition 11 separately.

(B1) Assignment:
Suppose, $\mathrm{Stmt}(l^{Iq}_n) = (u := f(u_1, \ldots, u_k), \bar{l})$ for some $u, u_1, \ldots, u_k, f, \bar{l}$. Then

$$l^{Iq}_{n+1} = \bar{l} \text{ and } \sigma^{Iq}_{n+1} \circ \sigma^{Iq}_n(u) = f^I(\sigma^{Iq}_n \circ \sigma^{Iq}_n(u_1), \ldots, \sigma^{Iq}_n \circ \sigma^{Iq}_n(u_k))\,. \tag{55}$$

By induction hypothesis,

$$l^z_n = l^{Iq}_n \ \text{ for sufficiently long } \ z \sqsubseteq q\,. \tag{56}$$

Since $J(u) = I(u) = \mathbf{Pointer}$ and $\sigma^{Iq}_n(u)$ exists, by induction hypothesis

$$v_u := \sigma^{Iq}_n(u) = \sigma^z_n(u) \text{ for sufficiently long } z \sqsubseteq q\,. \tag{57}$$

For $i = 1, \ldots, k$, $\sigma^{Iq}_n \circ \sigma^{Iq}_n(u_i)$ exists by (55), $w_i := \sigma^{Iq}_n(u_i) \in \mathrm{Var}$ and since $J(u_i) = I(u_i) = \mathbf{Pointer}$, by induction hypothesis,

$$\sigma^z_n(u_i) = \sigma^{Iq}_n(u_i) \text{ for sufficiently long } z \sqsubseteq q\,. \tag{58}$$

Since $I(w_i) = \boldsymbol{\Sigma^\omega}$, by induction hypothesis

$$x \mapsto \sigma^x_n(w_i) \ \text{ is monotone and } \ T_\omega(x \mapsto \sigma^x_n(w_i)) \ \text{ extends } \ p \mapsto \sigma^{Ip}_n(w_i) \tag{59}$$

for $i = 1, \ldots, k$.

Suppose, $I(v_u) \in \{\mathbf{Pointer}, \mathbf{Bool}\}$.

Then $f^J :\subseteq (\Sigma^*)^k \to \Sigma^*$ is monotone-constant such that $T_*(f^J)$ extends $f^I :\subseteq (\Sigma^\omega)^k \to \Sigma^*$ by assumption on I and J. By (59) and Lemma 5,

$$T_*(x \mapsto f^J(\sigma^x_n(w_1), \ldots, \sigma^x_n(w_k)) \ \text{ extends } \ p \mapsto f^I(\sigma^{Ip}_n(w_1), \ldots, \sigma^{Ip}_n(w_k))\,. \tag{60}$$

Hence, for sufficiently long $z \sqsubseteq q$,

$$f^J(\sigma_n^z(w_1),\ldots,\sigma_n^z(w_k)) = f^I(\sigma_n^{Iq}(w_1),\ldots,\sigma_n^{Iq}(w_k)) \tag{61}$$

and therefore by (56-58) for sufficiently long $z \sqsubseteq q$,

$$l_n^z = l_n^{Iq},\quad \sigma_n^z(u) = \sigma_n^{Iq}(u) \ \text{ and } \ f^J(\sigma_n^z \circ \sigma_n^z(u_1),\ldots,\sigma_n^z \circ \sigma_n^z(u_k)) \ \text{ exists}\,. \tag{62}$$

Therefore, for sufficiently long $z \sqsubseteq q$, $\mathrm{Stmt}(l_n^z) = (u := f(u_1,\ldots,u_k),\bar{l})$ and $(l_{n+1}^z,\sigma_{n+1}^z)$ exists.

In particular $l_{n+1}^z = \bar{l} = l_{n+1}^{Iq}$ for sufficiently long $z \sqsubseteq q$, if l_{n+1}^{Iq} exists.
Therefore, Property (52) is true for $I(v_u) \in \{\mathbf{Pointer},\mathbf{Bool}\}$.

We verify (53) and (54) for $I(v_u) \in \{\mathbf{Pointer},\mathbf{Bool}\}$.
Assume that $\sigma_{n+1}^{Iq}(v)$ exists.
(a1) Suppose, $I(v) \neq \boldsymbol{\Sigma^\omega}$, $v \neq v_u$.
Then $\sigma_{n+1}^{Iq}(v) = \sigma_n^{Iq}(v)$ and $\sigma_{n+1}^z(v) = \sigma_n^z(v)$ for sufficiently long $z \sqsubseteq q$.
By induction hypothesis, $\sigma_n^z(v) = \sigma_n^{Iq}(v)$ for sufficiently long $z \sqsubseteq q$. Therefore, $\sigma_{n+1}^z(v) = \sigma_{n+1}^{Iq}(v)$ for sufficiently long $z \sqsubseteq q$.
(a2) Suppose, $I(v) \neq \boldsymbol{\Sigma^\omega}$, $v = v_u$.
Then by (61, 62) for sufficiently long $z \sqsubseteq q$,

$$\begin{aligned}\sigma_{n+1}^z(v_u) &= \sigma_{n+1}^z \circ \sigma_n^z(u)\\ &= f^J(\sigma_n^z \circ \sigma_n^z(u_1),\ldots,\sigma_n^z \circ \sigma_n^z(u_k))\\ &= f^I(\sigma_n^{Iq} \circ \sigma_n^{Iq}(u_1),\ldots,\sigma_n^{Iq} \circ \sigma_n^{Iq}(u_k))\\ &= \sigma_{n+1}^{Iq} \circ \sigma_n^{Iq}(u)\\ &= \sigma_{n+1}^{Iq}(v_u)\,.\end{aligned}$$

Therefore, $\sigma_{n+1}^z(v_u) = \sigma_{n+1}^{Iq}(v_u)$ for sufficiently long $z \sqsubseteq q$.
(a3) Suppose, $I(v) = \boldsymbol{\Sigma^\omega}$.
Since by assumption $I(v_u) \in \{\mathbf{Pointer},\mathbf{Bool}\}$, $v \neq v_u$ and therefore as in (a1), $\sigma_{n+1}^{Iq}(v) = \sigma_n^{Iq}(v)$ and $\sigma_{n+1}^z(v) = \sigma_n^z(v)$ for sufficiently long $z \sqsubseteq q$.
Consider $j \in \mathbb{N}$. By induction hypothesis $\sigma_n^z(v) \sqsubseteq \sigma_n^{Iq}(v)$ and $|\sigma_n^z(v)| \geq j$ for sufficiently long $z \sqsubseteq q$. Therefore, $\sigma_{n+1}^z(v) \sqsubseteq \sigma_{n+1}^{Iq}(v)$ and $|\sigma_{n+1}^z(v)| \geq j$ for sufficiently long $z \sqsubseteq q$.
Therefore, Properties (53) and (54) are true for $I(v_u) \in \{\mathbf{Pointer},\mathbf{Bool}\}$.

Suppose $I(v_u) = \boldsymbol{\Sigma^\omega}$.

Then $f^J :\subseteq (\Sigma^*)^k \to \Sigma^*$ is monotone such that $T_\omega(f^J)$ extends $f^I :\subseteq (\Sigma^\omega)^k \to \Sigma^\omega$ by assumption on I and J. By Lemma 5 and (59),

$$T_\omega(x \mapsto f^J(\sigma_n^x(w_1),\ldots,\sigma_n^x(w_k))) \ \text{ extends } \ p \mapsto f^I(\sigma_n^{Ip}(w_1),\ldots,\sigma_n^{Ip}(w_k))\,. \tag{63}$$

Therefore by (56-58), for sufficiently long $z \sqsubseteq q$,

$$l_n^z = l_n^{Iq}, \quad \sigma_n^z(u) = \sigma_n^{Iq}(u) \text{ and } f^J(\sigma_n^z \circ \sigma_n^z(u_1), \ldots, \sigma_n^z \circ \sigma_n^z(u_k)) \text{ exists}. \tag{64}$$

And for sufficiently long $z \sqsubseteq q$, $\mathrm{Stmt}(l_n^z) = (u := f(u_1, \ldots, u_k), \bar{l})$ and $(l_{n+1}^z, \sigma_{n+1}^z)$ exists. In particular $l_{n+1}^z = \bar{l} = l_{n+1}^{Iq}$ for sufficiently long $z \sqsubseteq q$, if l_{n+1}^{Iq} exists. Therefore, Property (52) is true for $I(v_u) = \boldsymbol{\Sigma^\omega}$.

We verify (53) and (54) for $I(v_u) = \boldsymbol{\Sigma^\omega}$. Assume that $\sigma_{n+1}^{Iq}(v)$ exists.
(b1) Suppose, $I(v) \neq \boldsymbol{\Sigma^\omega}$, $v \neq v_u$.
The arguments are the same as in (a1) above.
(b2) Suppose, $I(v) = \boldsymbol{\Sigma^\omega}$, $v = v_u$.
Consider $j \in \mathbb{N}$. Then by (63, 64) for sufficiently long $z \sqsubseteq q$, $|\sigma_{n+1}^z(v_u)| \geq j$ and

$$\begin{aligned}
\sigma_{n+1}^z(v_u) &= \sigma_{n+1}^z \circ \sigma_n^z(u) \\
&= f^J(\sigma_n^z \circ \sigma_n^z(u_1), \ldots, \sigma_n^z \circ \sigma_n^z(u_k)) \\
&\sqsubseteq f^I(\sigma_n^{Iq} \circ \sigma_n^{Iq}(u_1), \ldots, \sigma_n^{Iq} \circ \sigma_n^{Iq}(u_k)) \\
&= \sigma_{n+1}^{Iq} \circ \sigma_n^{Iq}(u) \\
&= \sigma_{n+1}^{Iq}(v_u)\,.
\end{aligned}$$

Therefore, $\sigma_{n+1}^z(v_u) \sqsubseteq \sigma_{n+1}^{Iq}(v_u)$ and $|\sigma_{n+1}^z(v_u)| \geq j$ for sufficiently long $z \sqsubseteq q$.
(b3) Suppose, $I(v) = \boldsymbol{\Sigma^\omega}$ $v \neq v_u$.
The arguments are the same as in (a3) above.
Therefore, Properties (53) and (54) are true for $I(v_u) = \boldsymbol{\Sigma^\omega}$.

(B2) Branching:
Suppose, $\mathrm{Stmt}(l_n^{Iq}) = (\text{if } u \text{ then } l', l'')$ for some u, l', l''.
Then $\sigma_n^{Iq} \circ \sigma_n^{Iq}(u)$ exists,

$$l_{n+1}^{Iq} = \begin{cases} l' \text{ if } \sigma_n^{Iq} \circ \sigma_n^{Iq}(u) = \mathsf{1} \\ l'' \text{ if } \sigma_n^{Iq} \circ \sigma_n^{Iq}(u) = \mathsf{0} \end{cases}$$

and $\sigma_{n+1}^{Iq} = \sigma_n^{Iq}$. Since $u \in \mathrm{Var}$ and $\sigma_n^{Iq}(u) \in \mathrm{Var}$, by induction hypothesis $l_n^z = l_n^{Iq}$, $\sigma_n^z(u) = \sigma_n^{Iq}(u)$ and $\sigma_n^z(u) \circ \sigma_n^{Iq}(u) = \sigma_n^{Iq} \circ \sigma_n^{Iq}(u)$ and therefore, $\sigma_n^z(u) \circ \sigma_n^z(u) = \sigma_n^{Iq} \circ \sigma_n^{Iq}(u)$ for sufficiently long $z \sqsubseteq q$. Consequently for sufficiently long $z \sqsubseteq q$, $(l_{n+1}^z, \sigma_{n+1}^z)$ exists and

$$\begin{aligned}
l_{n+1}^{Iq} &= \begin{cases} l' \text{ if } \sigma_n^{Iq} \circ \sigma_n^{Iq}(u) = \mathsf{1} \\ l'' \text{ if } \sigma_n^{Iq} \circ \sigma_n^{Iq}(u) = \mathsf{0} \end{cases} && (65) \\
&= \begin{cases} l' \text{ if } \sigma_n^z \circ \sigma_n^z(u) = \mathsf{1} \\ l'' \text{ if } \sigma_n^z \circ \sigma_n^z(u) = \mathsf{0} \end{cases} && (66) \\
&= l_{n+1}^z\,. && (67)
\end{aligned}$$

Therefore, (52) is true. Since $\sigma^{Iq}_{n+1} = \sigma^{Iq}_n$ and $\sigma^z_{n+1} = \sigma^z_n$ for sufficiently long $z \sqsubseteq q$, Properties (53) and (54) can be proved as in (a1) and (a3) above.

Thus the proof of Proposition 36 is finished. $\square$ (Proposition 36)

We prove Lemma 14 for the case $f^I_F :\subseteq (\boldsymbol{\Sigma^\omega})^k \to \boldsymbol{\Sigma^\omega}$.
For the case $f^I_F :\subseteq (\boldsymbol{\Sigma^\omega})^k \to Y$, $Y \in \{\mathbf{Pointer}, \mathbf{Bool}\}$, the proof is similar.
First, we show that f^J_F in monotone. Assume that $f^J_F(y)$ exists and suppose $y \sqsubseteq z$. By Definition 11 there is some n such that $(l^y_n, \sigma^y_n(v_0)) = (l_{fin}, f^J_F(y))$. By 36(1), $l^y_n = l^z_n$, and by 36(3), $\sigma^y_n(v_0) \sqsubseteq \sigma^z_n(v_0)$ since $J(v_0) = \boldsymbol{\Sigma}^*$. Therefore, $(l^z_n, \sigma^z_n(v_0)) = (l_{fin}, \sigma^z_n(v_0))$ and so $f^J_F(z) = \sigma^z_n(v_0)$. We obtain $f^J_F(y) \sqsubseteq f^J_F(z)$.

Next, we show that $T_\omega(f^J_F)$ extends f^I_F. Suppose $f^I_F(q)$ exists, let $j \in \mathbb{N}$. There is some n such that $(l^{Iq}_n, \sigma^{Iq}_n(v_0)) = (l_{fin}, f^I_F(q))$. By 36(1) $l^z_n = l^{Iq}_n$ for sufficiently long $z \sqsubseteq q$. Since $I(v_0) = \boldsymbol{\Sigma^\omega}$, by 36(3), $\sigma^z_n(v_0) \sqsubseteq \sigma^{Iq}_n(v_0)$ and $|\sigma^z_n(v_0)| \geq j$ for sufficiently long $z \sqsubseteq q$. Therefore, for sufficiently long $z \sqsubseteq q$, $(l^z_n, \sigma^z_n) = (l_{fin}, \sigma^z_n)$, hence $f^J_F(z) = \sigma^z_n(v_0) \sqsubseteq \sigma^{Iq}_n(v_0) = f^I_F(q)$ and $|f^J_F(z)| \geq j$. Hence, $T_\omega(f^J_F)$ extends f^I_F.

B Proof of Theorem 23

For $\gamma_v : \tau_v \rightrightarrows \tau'_v$ let U_v and X_v be the sets underlying the types τ_v and τ'_v, respectively. Hence we can write $\gamma_v : U_v \rightrightarrows X_v$. Let $v_1, \ldots, v_m$ the input registers and let v_0 the output register of the flowchart scheme F (Definition 9). For $\bar{u} \in U_{v_1} \times \ldots \times U_{v_m}$, $w \in U_{v_0}$ and $\bar{z} \in X_{v_1} \times \ldots \times X_{v_m}$ by (17) we must show:

$$\text{if} \quad \bar{z} \in (\gamma_{v_1} \times \ldots \times \gamma_{v_m})(\bar{u}) \cap \mathrm{dom}(f^I_F) \tag{68}$$

$$\text{then } f^J_F(\bar{u}) \neq \emptyset \quad \text{and} \quad \forall w \in f^J_F(\bar{u}).\, f^I_F(\bar{z}) \cap \gamma_{v_0}(w) \neq \emptyset\,. \tag{69}$$

By (39) for each variable v, $I(v) = J(v) = \mathbf{Pointer}$, if $\gamma_v = \gamma^{var}$ and $I(v) = J(v) = \mathbf{Bool}$, if $\gamma_v = \gamma^{bool}$. Let us say that the state σ of (F, J) realizes the state σ' of (F, I), if

$$(\forall\, v \in \mathrm{Var}) \;\; \sigma'(v) \in \gamma_v \circ \sigma(v) \;\; \text{if} \;\; \sigma'(v) \;\; \text{exists.} \tag{70}$$

In Figure 6, σ realizes σ'. If σ realizes σ' then

$$\sigma(v) = \sigma'(v) \;\; \text{if} \;\; \gamma_v \in \{\gamma^{var}, \gamma^{bool}\} \quad \text{and} \;\; \sigma'(v) \text{ exists.} \tag{71}$$

The following proposition generalizes (17) in Definition 17.

Proposition 37. *Let (l, σ) be a configuration of the flowchart $(F, J$ and let (l, σ') be a configuration of the flowchart $(F; I)$ such that σ realizes σ' and (l, σ') has a $\overset{I}{\vdash}$ -successor.*

Then (l, σ) has a $\overset{J}{\vdash}$ -successor and for all $(\bar{l}, \bar{\sigma})$ such that $(l, \sigma) \overset{J}{\vdash} (\bar{l}, \bar{\sigma})$ there is some $\bar{\sigma}'$ such that $(l, \sigma') \overset{I}{\vdash} (\bar{l}, \bar{\sigma}')$ and $\bar{\sigma}$ realizes $\bar{\sigma}'$.

Proof: (Proposition 37) Since (l,σ') has a $\stackrel{I}{\longmapsto}$-successor, Stmt$(l)$, which is either an assignment or a branching, can be applied to the configuration σ'.

Assignment: Suppose $\mathrm{Stmt}(l) = (u := f(u_1,\dots,u_k),\bar{l})$ for some f and $u,u_1,\dots,u_k \in \mathrm{Var}$. Then by (8), $f^I(\sigma'\circ\sigma'(u_1),\dots,\sigma'\circ\sigma'(u_k)) \neq \emptyset$. Since σ realizes σ' by (71) there are variables $t,t_1,,\dots,t_k$ such that

$$t=\sigma'(u)=\sigma(u),\ t_1=\sigma'(u_1)=\sigma(u_1),\ \dots,\ t_k=\sigma'(u_k)=\sigma(u_k).$$

By By Assumption (40) and Condition 2 from Theorem 23, generalized multi-representations $\gamma_{t_1},\dots,\gamma_{t_k},\gamma_t \in \mathrm{GR}$ are uniquely determined by the type of f^I such that f^J is a $(\gamma_{t_1}\times\dots\times\gamma_{t_k},\gamma_t)$-realization of f^I. Therefore, if

$$(\sigma'(t_1),\dots,\sigma'(t_k)) \in (\gamma_{t_1}\times\dots\times\gamma_{t_k})(\sigma(t_1),\dots,\sigma(t_k)) \text{ and} \tag{72}$$
$$(\sigma'(t_1),\dots,\sigma'(t_k)) \in \mathrm{dom}(f^I) \tag{73}$$

then
$$f^J(\sigma(t_1),\dots,\sigma(t_k)) \neq \emptyset \quad \text{and} \tag{74}$$
$$(\forall w \in f^J(\sigma(t_1),\dots,\sigma(t_k)))\ \ f^I(\sigma'(t_1),\dots,\sigma'(t_k)) \cap \gamma_t(w) \neq \emptyset \tag{75}$$

(72) is true since σ realizes σ', (73) is true since (l,σ') has a successor. Therefore, (74, 75) are true. Since $f^J(\sigma(t_1),\dots,\sigma(t_k)) = f^J(\sigma\circ\sigma(u_1),\dots,\sigma\circ\sigma(u_k))$, by (74) (l,σ) has a $\stackrel{J}{\longmapsto}$-successor.

Let $(\tilde{l},\bar{\sigma})$ be a $\stackrel{J}{\longmapsto}$-successor of (l,σ). By the form of the statement of l, $\tilde{l}=\bar{l}$. The configuration $\bar{\sigma}$ differs from σ (at most) for the variable $t=\sigma(u)=\sigma'(u)$ and any next configuration of σ' differs from σ' (at most) for the variable $t=\sigma(u)=\sigma'(u)$. So by (70) for the new value $\bar{\sigma}(t)\in f^J(\sigma\circ\sigma(u_1),\dots,\sigma\circ\sigma(u_k))$ it suffices to find some new value $\bar{\sigma}'(t)$ such that $\bar{\sigma}'(t)\in\gamma_t\circ\bar{\sigma}(t)$ and $\bar{\sigma}'(t)\in f^I(\sigma'(t_1),\dots,\sigma'(t_k))$. The existence of such a value is guaranteed by (75).

Branching: Suppose $\mathrm{Stmt}(l) = (\text{if } u \text{ then } l',l'')$. Since a $\stackrel{I}{\longmapsto}$-successor exists, $t:=\sigma'(u)\in\mathrm{Var}$ exists and $\sigma'(t)\in\mathrm{Bool}$ exists by Definition 11(2). From (71), $\sigma(u)=\sigma'(u)$ and $\sigma(t)=\sigma'(t)$. Therefore, the unique successor of (l,σ) is (l',σ) or (l'',σ) if the unique successor of (l,σ') is (l',σ') or (l'',σ'), respectively. $\square$(Proposition 37)

Assume (68), i.e., $\bar{z}\in(\gamma_{v_1}\times\dots\times\gamma_{v_m})(\bar{u})\cap\mathrm{dom}(f_F^I)$. We must prove (69). Let (l_0,σ_0) and (l_0,σ_0') be the initial configurations of F with interpretation J and input $\bar{u}$ and of F with interpretation I and input $\bar{z}$, respectively. Then σ_0 realizes σ_0'. The values $f_F^I(\bar{u})$ and $f_F^J(\bar{z})$ are defined by means of *acceptable* computations, see Definition 11.

Proposition 38. *There is an acceptable $\overline{u}$-computation for (F,J).*

Proof: (Proposition 38) Define inductively sequences $(l_0,\sigma_0),(l_1,\sigma_1),\dots$ and $(l_0,\sigma_0'),(l_1,\sigma_1'),\dots$ such that $(l_i,\sigma_i)\stackrel{J}{\longmapsto}(l_{i+1},\sigma_{i+1})$, $(l_i,\sigma_i')\stackrel{I}{\longmapsto}(l_{i+1},\sigma_{i+1}')$ and σ_{i+1} realizes σ_{i+1}' as follows:

Suppose, $l_i \neq l_{fin}$ (the final label). Since $\bar{z} \in \mathrm{dom}(f_F^I)$ and $l_i \neq l_{fin}$, the $\bar{z}$-computation $(l_0, \sigma_0'), \ldots, (l_i, \sigma_i')$ cannot be maximal by (16) and hence (l_i, σ_i') has a $\stackrel{I}{\vdash}$ -successor. By Proposition 37, (l_i, σ_i) has a $\stackrel{J}{\vdash}$ -successor (l_{i+1}, σ_{i+1}) and there is some $\stackrel{I}{\vdash}$ -successor (l_{i+1}, σ_{i+1}') of (l_i, σ_i') such that σ_{i+1} realizes σ_{i+1}'.

If $l_i = l_{fin}$ (the final label) then (l_i, σ_i) and (l_i, σ_i') are the last elements of the sequences.

Let $(l_0, \sigma_0'), (l_1, \sigma_1'), \ldots$ be the maximal computation constructed in this way. Since it is a $\bar{z}$-computation and $\bar{z} \in \mathrm{dom}(f_F^I)$, by (16) it must be acceptable and therefore has a last element (l_{fin}, σ_n') with defined result $\sigma_n'(v_0) \in f_F^I(\bar{z})$. Therefore, $(l_0, \sigma_0), (l_1, \sigma_1), \ldots$ has a last element (l_{fin}, σ_n). Since σ_n realizes σ_n', by (70) $\sigma_n'(v_0) \in \gamma_{v_0} \circ \sigma_n(v_0)$. Therefore, $\sigma_n'(v_0)$ exists and hence we have an acceptable $\bar{u}$-computation for (F, J). $\square$(Proposition 38)

Proposition 39. *For every $\bar{u}$-computation $(l_0, \sigma_0), \ldots, (l_n, \sigma_n)$ in (F, J) there is a $\bar{z}$-computation $(l_0, \sigma_0'), \ldots, (l_n, \sigma_n')$ in (F, I) such that σ_i realizes σ_i' for $i = 1, \ldots, n$.*

Proof: (Proposition 39) This is true for $n = 0$. Let $(l_0, \sigma_0), \ldots, (l_n, \sigma_n)$, (l_{n+1}, σ_{n+1}) be a $\bar{u}$-computation. By induction hypothesis there is a $\bar{z}$-computation $(l_0, \sigma_0'), \ldots, (l_n, \sigma_n')$ such that σ_i realizes σ_i' for $i = 1, \ldots, n$. Since (l_n, σ_n) has a $\stackrel{J}{\vdash}$ -successor, l_n is not the final label l_{fin}. Since $\bar{z} \in \mathrm{dom}(f_F^I)$, every maximal $\bar{z}$-computation is acceptable, hence (l_n, σ_n') has a $\stackrel{I}{\vdash}$ -successor. By Proposition 37, (l_n, σ_n') has a $\stackrel{I}{\vdash}$ -successor (l_{n+1}, σ_{n+1}') such that σ_{n+1} realizes σ_{n+1}'. $\square$(Proposition 39)

Proposition 40. *Every maximal $\bar{u}$-computation in (F, J) is acceptable.*

Proof: (Proposition 40) Let $S := (l_0, \sigma_0), \ldots, (l_n, \sigma_n)$ be a maximal $\bar{u}$-computation. Let $S' := (l_0, \sigma_0'), \ldots, (l_n, \sigma_n')$ be some $\bar{z}$-computation according to Proposition 39. Suppose, $l_n \neq l_{fin}$. Since every maximal $\bar{z}$-computation is acceptable, this computation cannot be maximal and must have an $\stackrel{I}{\vdash}$ -successor. By Proposition 37 (l_n, σ_n) must have a $\stackrel{J}{\vdash}$ -successor (contradiction). Therefore, $l_n 0 l_{fin}$. Then S' is a maximal $\bar{z}$-computation, $\sigma_n'(v_0)$ exists. Since by Proposition 39, σ_n realizes σ_n', $\sigma_n(v_0)$ exists. Therefore, the computation S is acceptable. $\square$(Proposition 40)

Since there is an acceptable $\bar{u}$-computation and every maximal $\bar{u}$-computation is acceptable, by (16) $f_F^J(\bar{u}) \neq \emptyset$, which is the first part of (69). It remains to show $\forall w \in f_F^J(\bar{u}).\, f_F^I(\bar{z}) \cap \gamma_{v_0}(w) \neq \emptyset$, the second part of (69).

Suppose, $w \in f_F^J(\bar{u})$. Then there is an acceptable $\bar{u}$-computation $S := (l_0, \sigma_0), \ldots, (l_n, \sigma_n)$ such that $w = \sigma_n(v_0)$. By Proposition 39 there is a $\bar{z}$-computation $S' := (l_0, \sigma'_0), \ldots, (l_n, \sigma'_n)$ such that σ_n realizes σ'_n. Since S is acceptable, $l_n = l_{fin}$. Then S' is a maximal $\bar{z}$-computation which must be acceptable by (16), hence $\sigma'_n(v_0) \in f_F^I(\bar{z})$. Since σ_n realizes σ'_n, $\sigma'_n(v_0) \in \gamma_{v_0} \circ \sigma_n(v_0)$. We obtain $\sigma'_n(v_0) \in f_F^I(\bar{z}) \cap \gamma_{v_0}(w)$, hence $f_F^I(\bar{z}) \cap \gamma_{v_0}(w) \neq \emptyset$. Therefore, (69) has been proved.