Noname manuscript No.
(will be inserted by the editor)

On Two Notions of Computation in Transparent Intensional Logic

Received: date / Accepted: date

Abstract In Transparent Intensional Logic developed by Pavel Tichý can be recognized two distinct notions of computation that loosely correspond to term rewriting and term interpretation as known from lambda calculus. Our goal will be to further explore these two notions and examine some of their properties.

Keywords transparent intensional logic · procedural semantics · lambda calculus · term rewriting · term interpretation

1 Introduction and motivation

Transparent Intensional Logic (TIL) developed by Pavel Tichý (see Tichý (1988)[1]) provides a procedural semantics for natural language analysis. Formally, it is based on typed lambda calculus with partial functions, whose terms are interpreted procedurally: λ-terms are taken as denoting certain abstract procedures, so called *constructions*, that can be executed. For example, the following construction

$$[[\lambda x\, x]\, \mathbf{1}]$$

funding

Address(es) of author(s) should be given

[1] For more about TIL, see Duží et al. (2010), Raclavský et al. (2015) or Materna (2013) and Sierszulska (2007) in this journal.

is not understood just as a λ-term that can be β-reduced

$$[[\lambda x\, x]\, \mathbf{1}] \longrightarrow_{\beta} \mathbf{1}$$

into its normal form **1**, but rather as an instruction to apply an identity function to the number 1.[2] Naturally, such a procedure yields as its output the number 1. In TIL terminology, we would say that the construction $[[\lambda x\, x]\, \mathbf{1}]$ constructs the number 1. Of course, if we take $[[\lambda x\, x]\, \mathbf{1}]$ as an abstract procedure it doesn't mean it ceases to be open to syntactic operations: it can still be β-reduced into **1**. The key point is that depending on which route we take with $[[\lambda x\, x]\, \mathbf{1}]$ – the semantic one or the syntactic one – we get different outcomes. The semantic route returns the number 1, the syntactic route delivers the construction **1**, which can be regarded as a trivial construction constructing the number 1 again.

These two routes correspond to two notions of computation present in TIL: syntactic computation (β-reduction is commonly understood as capturing the notion of a computational step[3]) and semantic computation (TIL constructions ought to represent computations[4]). The presence of these two routes, which roughly correspond to term rewriting and term interpretation from lambda calculus, is generally known in TIL (see e.g., Raclavský et al. (2015), p. 255), however, not often directly discussed and studied. The main of this paper will be to examine these two notions of computation in more detail.

This paper is organized as follows: in Section 2 we introduce the basics of TIL. In Section 3 we examine the semantic and the syntactic notions of computation in TIL (including their potential reducibility) and in Section 4 we explore the possible interaction between these two notions.

[2] Note the difference between **1** and 1, which will be explained below.

[3] See e.g., Barendregt (2013).

[4] Tichý (1988) often likens constructions to calculations (see e.g., pages 7, 12, 20, 31, 82, 222, 281 in Tichý (1988)). The more general term *computation*, however, can be no doubt used as well, considering Tichý 'calculates' also truth values, individuals, etc.

2 Transparent intensional logic: brief overview

2.1 Setting the stage

Let us have a look at the following expression:

$$5+7=12.$$

It is an equality statement and as such it should hold between objects of the same type. There are essentially two (non-syntactic) ways we can interpret it based on the reading direction:

- *from right to left*: since 12 is a number and it is equal to $5+7$, then $5+7$ must be a number as well;
- *from left to right*: since $5+7$ is a calculation (procedure, construction, ...) and it is equal to 12, then 12 must be a calculation as well.

Of course, neither of these two options is satisfactory. It seems a little peculiar to say that $5+7$ is a number. Our immediate reaction would probably be that it is rather some sort of calculation leading to a number. But at the same time it seems odd to say that 12 is a calculation. How do we calculate it? Isn't it rather the result of some calculation?[5] Both approaches bring their own set of challenges and advantages and we do not intent to settle the question "who's right?" here. The purpose of this example is simply to set the stage for TIL, which explores the *from left to right* reading. In TIL, expressions $5+7$ and 12 will be understood as expressing certain constructions, informally, constructions "apply the addition function to the numbers 5 and 7" and "take the number 12", respectively.[6]

[5] For analogous observations, see e.g., Tichý (1988), Girard et al. (1989).

[6] The *from left to right* interpretation is also followed by Moschovakis (2006), Muskens (2005), Girard et al. (1989). The *from right to left* interpretation is explored e.g., in Martin-Löf's constructive type theory (CTT, see Martin-Löf 1984). For a comparison between CTT and TIL, see e.g., [blinded], Primiero and Jespersen (2010).

2.2 Constructions

In our presentation of TIL, we will rely on four basic kinds of constructions: variable, composition, closure, and n-execution. They are captured by the following syntax:

construction	*notation*
variable	x
composition	$[C\,C_1 \ldots C_m]$
closure	$[\lambda x_1 \ldots x_m\, C]$
n-execution	${}^{n}X$

where C_i is any construction and X is an arbitrary construction or a non-construction (e.g., truth value, number, etc.).

The first three constructions, i.e., variable, composition, and closure, correspond roughly to variable, function application, and function abstraction as known from lambda calculus. In many circumstances they can behave in the same way (e.g., both λ-terms and constructions are open to α-, β-, η-conversions). There are, however, important conceptual and practical differences between them. Most notably, constructions of TIL are not terms but abstract objects that can be carried out to construct other objects. This brings us to the last construction called n-execution, which exemplifies this procedural behaviour of constructions.[7] Executions can be thought of as procedures of running the corresponding constructions. For example, assume that X is some construction C constructing an object a, then ${}^{0}C$ can be interpreted as an instruction not to execute the construction C,[8] construction ${}^{1}C$ can be seen as an instruction to execute C, ${}^{2}C$ as an instruction to execute C and then also execute its result a (which can be also another construction) and so on.[9]

Each construction C constructs an object with respect to some *valuation* v, which is a function assigning values to variables from a sequence of objects of certain

[7] Traditionally, TIL is equipped with single execution, double execution and trivialisation construction. The notion of n-execution is a generalization of these constructions, see Appendix 6.

[8] Zero execution has many uses in TIL, one of them being introduction of pre-defined objects into TIL constructions. From this perspective, we can view zero executions as analogous to constants in impure lambda calculus.

[9] For a proper specification of all constructions, see Appendix 6.

type. Hence we say that constructions v-*construct* objects. For example, we will say that construction x v-constructs the number 12 (assuming v assigns 12 to x), ${}^0 1$ v-constructs the number 1 (for any valuation), $[{}^0\mathrm{Add}\ {}^0 5\ {}^0 7]$ v-constructs 12 (again, for any valuation), $[{}^0{=}\ [{}^0\mathrm{Add}\ {}^0 5\ {}^0 7]\ {}^0 11]$ v-constructs $\mathtt{F}$ (i.e., falsehood), etc. If a construction C v-constructs nothing at all, we will say that C is a v-*improper* construction. Otherwise, we say that it is a v-*proper* construction. Constructions are called v-*congruent* (symbolized by $\cong$), if they both v-construct the same object, or they are both v-improper. Constructions are called *logically equivalent* (symbolized by $\equiv$), if they are v-congruent for any v.

Remark 1 More generally, we can think of constructions as explications of Frege's notion of *sense*, their results being the corresponding denotations.[10] In other words, constructions are understood as meanings of natural language expressions. In that respect, TIL falls under the propositions-as-algorithms paradigm.[11]

To simplify the notation, we will symbolize zero execution by **boldface** font with the exception of equality sign. We will also use infix notation whenever convenient. Thus, e.g., ${}^0 1$ becomes $\mathbf{1}$, $[{}^0{=}\ [{}^0\mathrm{Add}\ {}^0 5\ {}^0 7]\ {}^0 11]$ becomes $[[\mathbf{Add}\ \mathbf{5}\ \mathbf{7}] = \mathbf{11}]$, etc.

3 Two notions of computation

In TIL we can recognize two notions of computation of constructions, which we will call syntactic and semantic:

i. syntactic computation ('construction transformation'): corresponds to what is meant by computation in lambda calculus, i.e., *computational step* coincides with β-*reduction*; e.g., the step from the construction $[[\lambda x\, x]\, \mathbf{1}]$ to the construction $\mathbf{1}$,
ii. semantic computation ('v-constructing'): roughly correlates to what is meant by interpretation in lambda calculus, i.e., *computational step* coincides with a specific interpretation of a λ-term; e.g., the step from the construction $[[\lambda x\, x]\, \mathbf{1}]$ to the non-construction 1.

[10] More accurately, they aim to explicate Frege's notion of *conceptual content* from *Begriffschrift* (see e.g., van Heijenoort 1977). For a more detailed examination, see Tichý (1988).
[11] See also Jespersen (2017), Muskens (2005), Moschovakis (2006).

Schematically, assume that C and D are proper constructions. Furthermore, assume that C is a β-reducible construction, D is a β-reduced construction from C, and α is an object ν-constructed by both C and D, then (see Figure 1):

$$\begin{tikzcd}
C \arrow[rr, two heads, "\beta" description] \arrow[dr, two heads, "\nu" description] & & D \arrow[dl, two heads, "\nu" description] \\
& \alpha &
\end{tikzcd}$$

Fig. 1 Syntactic and semantic computations

where the β label represents the syntactic computational step (C is β-reducible to D) and the ν label denotes the semantic constructional steps (both C and D ν-construct the same object α).

Remark 2 We assume that β-reduction preserves strict equivalence between constructions, i.e., that both C and D either ν-construct the same object for every valuation ν, or they are both ν-improper. For more on the topic of β-conversion and TIL, see Duží (2017).

For example, consider the construction $[[\lambda x\, x]\, \mathbf{1}]$. Then we get (see Figure 2):

$$\begin{tikzcd}
{[[\lambda x\, x]\, \mathbf{1}]} \arrow[rr, two heads, "\beta" description] \arrow[dr, two heads, "\nu" description] & & \mathbf{1} \arrow[dl, two heads, "\nu" description] \\
& 1 &
\end{tikzcd}$$

Fig. 2 Syntactic and semantic computations of $[[\lambda x\, x]\, \mathbf{1}]$

In other words, $[[\lambda x\, x]\, \mathbf{1}]$ is β-reducible to **1** (syntactic computational step) and both $[[\lambda x\, x]\, \mathbf{1}]$ and **1** ν-construct the number 1 (semantic constructional steps). These two notions of computation can be defined as follows:

Definition 1 (*Syntactic computability*) Construction C will be said to be *syntactically computable* if there exists some construction C′ different from C that can be obtained by applying β-reduction to C. Otherwise, we say that C is syntactically not computable, i.e., that C has itself as value.

Definition 2 (*Semantic computability*) Construction C will be said to be *semantically computable* if there exists some object a that this construction v-constructs for some valuation v (in other words, if C is a v-proper for some v, i.e., if C is not an improper construction). Otherwise, we say that it is semantically not computable.

Instead of syntactic and semantic computability we will sometimes use the more specific terms β-computability and v-computability.

To better demonstrate the difference between syntactic and semantic computability, consider e.g., the following four constructions

i) $[[\lambda x[\lambda x[\lambda x[\mathbf{Succ}\, x]]x]x]\ \mathbf{0}]$
ii) $[[\lambda x[\lambda x[\mathbf{Succ}\, x]]x]\ \mathbf{0}]$
iii) $[[\lambda x[\mathbf{Succ}\, x]]\ \mathbf{0}]$
iv) $[\mathbf{Succ}\ \mathbf{0}]$

From the perspective of syntactic computation, i) computes into ii), ii) into iii), and finally iii) into iv). From the perspective of semantic computation, all these constructions compute the number 1 (schematically, see Figure 3). In other words, β-reducible constructions i)–iii) into iv) are understood as v-constructing the same object, i.e., the number 1. We can generalize this observation and say that we can have number $n-1$ ($n > 1$) of β-reducible (syntactically computable) proper constructions $A_1, \ldots, A_{n-1}$ into A_n, each one of them 'converging' via v-computability (semantic computation) into a, which is the object v-constructed by each A_i (schematically, see Figure 4).

Remark 3 Note that all the construction i)–iv) from the above case semantically compute 1 in a single computational step. For example, $[[\lambda x[\lambda x[\lambda x[\mathbf{Succ}\, x]]x]x]\ \mathbf{0}]$ and $[[\lambda x[\mathbf{Succ}\, x]]\ \mathbf{0}]$ directly v-construct/compute into 1, even though single syntactic computation (via β-reduction) reduces the former into $[[\lambda x[\mathbf{Succ}\, x]]\ \mathbf{0}]$ and the latter into $[\mathbf{Succ}\ \mathbf{0}]$, which is from the pure lambda calculus viewpoint further irreducible.

$$\begin{tikzcd}
 & & & & {[[\lambda x\,[\lambda x\,[\lambda x\,[\mathbf{Succ}\,x]]x]x]\,\mathbf{0}]} \arrow[dl, two heads, "\beta"'] \arrow[dr, two heads, "\nu"] & \\
 & & & {[[\lambda x\,[\lambda x\,[\mathbf{Succ}\,x]]x]\,\mathbf{0}]} \arrow[dl, two heads, "\beta"'] \arrow[dr, two heads, "\nu"] & & 1 \\
 & & {[[\lambda x\,[\mathbf{Succ}\,x]]\,\mathbf{0}]} \arrow[dl, two heads, "\beta"'] \arrow[dr, two heads, "\nu"] & & 1 & \\
 & {[\mathbf{Succ}\,\mathbf{0}]} \arrow[dl, two heads, "\cancel{\beta}"'] \arrow[dr, two heads, "\nu"] & & 1 & & \\
\ldots & & 1 & & &
\end{tikzcd}$$

Fig. 3 Parallel progress of syntactic and sematnic computational steps

$$\begin{tikzcd}
A_1 \arrow[r, two heads, "\beta"] \arrow[drrrr, two heads, "\nu"] & A_2 \arrow[r, two heads, "\beta"] \arrow[drrr, two heads, "\nu"] & \ldots \arrow[r, two heads, "\beta"] \arrow[drr, two heads, "\nu"] & A_{n-1} \arrow[r, two heads, "\beta"] \arrow[dr, two heads, "\nu"] & A_n \arrow[d, two heads, "\nu"] \\
 & & & & a
\end{tikzcd}$$

Fig. 4 Convergence of semantic/ν-computations

3.1 Translation between syntactic and semantic computation

Depending on what 'computation route' we take, we get different results. For example, as already discussed above, construction $[[\lambda x\,x]\,\mathbf{1}]$ β-computes to $\mathbf{1}$, but ν-computes to 1. Schematically (see Figure 5):

Even though the difference between syntactic and semantic computation might seem negligible in simple cases just as this one (after all, $\mathbf{1}$ 'directly' ν-constructs 1), there are other cases that exhibit much less straightforward behaviour. Consider e.g., the following four scenarios A, B, C, and D (see Figures 6, 7, 8, and 9).

$$\begin{tikzcd}
 & {[[\lambda x\, x]\, \mathbf{1}]} \arrow[dl, two heads, "\beta"', "\text{syntactic}"] \arrow[dr, two heads, "\nu", "\text{semantic}"'] & \\
\mathbf{1} & & 1
\end{tikzcd}$$

Fig. 5 Syntactic and semantic computational routes

Remark 4 By striked β, i.e., $\not\beta$, we will signify that β-reduction cannot be applied (= no constructions available for reduction). The ellipsis $\cdots$ will then stand for this further β-irreducible construction (essentially, a β-normal form) appearing in the preceding node. And finally, $\varnothing$ will denote nothing, i.e., the result of an improper construction.

$$\begin{tikzcd}
 & {[[\lambda x [\mathbf{Div}\ x\ \mathbf{0}]]\ \mathbf{5}]} \arrow[dl, two heads, "\beta"'] \arrow[dr, two heads, "\nu"] & \\
{[\mathbf{Div}\ \mathbf{5}\ \mathbf{0}]} \arrow[d, two heads, "\not\beta"'] \arrow[dr, two heads, "\nu"] & & \varnothing \\
\cdots & \varnothing &
\end{tikzcd}$$

Fig. 6 Case A

$$\begin{tikzcd}
 & \mathbf{1} \arrow[dl, two heads, "\not\beta"'] \arrow[dr, two heads, "\nu"] & \\
\cdots & & 1
\end{tikzcd}$$

Fig. 7 Case B

$$\begin{tikzcd}
 & {[[\lambda x [\mathbf{Add}\ \mathbf{5}\ x]]\ \mathbf{7}]} \arrow[dl, two heads, "\beta"'] \arrow[dr, two heads, "\nu"] & \\
{[\mathbf{Add}\ \mathbf{5}\ \mathbf{7}]} \arrow[d, two heads, "\not\beta"'] \arrow[dr, two heads, "\nu"] & & 12 \\
\cdots & 12 &
\end{tikzcd}$$

Fig. 8 Case C

$$\begin{tikzcd}
 & {[\mathbf{5}\ x]} \arrow[dl, two heads, "\not\beta"'] \arrow[dr, two heads, "\nu"] & \\
\cdots & & \varnothing
\end{tikzcd}$$

Fig. 9 Case D

As can be seen, we can get varying results:

A: [[λx[**Div** x **0**] **5**] is syntactically computable into [**Div 5 0**], but not semantically computable – it is so called improper construction, i.e., construction that fails to ν-construct anything.

B: **1** is semantically computable to 1, but syntactically non-computable since there is nothing to β-reduce.

C: [[λx[**Add 5** x]] **7**] is computable both syntactically and semantically: syntactic computation yields [**Add 5 7**], semantic one yields 12. Note, however, that the result is no longer both syntactically and semantically computable, see case B.

D: [**5** x] is not computable either syntactically or semantically: there is nothing to β-reduce and it is incorrectly formed composition construction (see definitions of constructions in Appendix 6).

Remark 5 Note that e.g., [x y], [**Add** x y], [z **5 7**], [z **5** y], [z x y] are all properly formed constructions of TIL that are potentially (i.e., depending on some valuation ν) semantically computable.[12]

Thus, as illustrated by cases A, B, C, and D, the results of syntactic and semantic computations are not always as similar as in our motivational example [[λx x] **1**]. Additionally, we can recognize the following four basic computational behaviours of constructions:

1. construction is both syntactically and semantically computable (case C above),
2. construction is syntactically but not semantically computable (case A above),
3. construction is semantically but not syntactically computable (case B above),
4. construction is neither semantically nor syntactically computable (case D above).

We will adopt the following metalanguage shorthands:

– symbol ■■ will represent constructions that are both β- and ν-computable,
– symbol ■□ will represent constructions that are β-computable but not ν-computable,
– symbol □■ will represent constructions that are ν-computable but not β-computable,
– symbol □□ will represent constructions that are neither β- nor ν-computable.

[12] Let us not confuse [x y] with “apply variable x to variable y”, the proper reading should be “apply the function ν-constructed by variable x to the argument ν-constructed by variable y”.

Informally, ■■, ■□, □■, and □□ can be understood as four basic computation types of constructions we can encounter in TIL. E.g., we can say that some construction A (e.g., **1** of type □■) has different computation type than construction B (e.g., $[[\lambda x\, x]\ \mathbf{1}]$ of type ■■). Probably the most interesting ones are the ■□ and □■ computation types since they represent constructions that are, in a way, both computable and non-computable. More specifically, they signify that we can have constructions that are syntactically computable but not semantically and vice versa.

To sum up, our analysis of the cases A, B, C, D indicates that the two notions of computations—syntactic and semantic—are not convertible into each other, since they can yield varying results. More formally, we can prove this in the following way. First, we need to introduce two new notions:

Definition 3 (β-*computation route*) Let A, a be constructions. We say that there is a β-*computation route* from A to a if A can be reduced to a via β-reduction modulo zero execution (written as $A \twoheadrightarrow^{0}_{\beta} a$).[13]

Definition 4 (ν-*computation route*) Let A be a construction and a either a construction or a non-construction. We say that there is a ν-*computation route* from A to a if A constructs a w.r.t. valuation ν (written as $A \twoheadrightarrow_{\nu} a$).

Remark 6 Note that $A \twoheadrightarrow^{0}_{\beta} a$ coincides with $A \longrightarrow_{\nu} a$ in basic cases, e.g., compare $[[\lambda x\, x]\ \mathbf{1}] \twoheadrightarrow^{0}_{\beta} 1$ and $[[\lambda x\, x]\ \mathbf{1}] \longrightarrow_{\nu} 1$.

Second, let us denote by $\Pi(\beta,\nu)$ the translatability of β-computation into ν-computation, by $\Pi(\nu,\beta)$ the translatability of ν-computation into β-computation, and finally by $\Pi(\mathtt{full})$ their intertranslatability. Thus

$$\begin{aligned}
\Pi(\beta,\nu) \quad &\text{iff} \quad \forall A \forall a (A \twoheadrightarrow^{0}_{\beta} a \implies A \twoheadrightarrow_{\nu} a)\\
\Pi(\nu,\beta) \quad &\text{iff} \quad \forall A \forall a (A \twoheadrightarrow_{\nu} a \implies A \twoheadrightarrow^{0}_{\beta} a)\\
\Pi(\mathtt{full}) \quad &\text{iff} \quad \forall A \forall a (A \twoheadrightarrow^{0}_{\beta} a \iff A \twoheadrightarrow_{\nu} a)
\end{aligned}$$

[13] By “modulo zero execution” we simply mean ignoring the outermost zero execution. For example, **12** modulo zero execution becomes 12 (recall that **12** is a shorthand for $^{0}12$).

In other words, we say that β-computation is translatable into ν-computation (i.e., $\Pi(\beta,\nu)$) iff it holds that for all constructions A and all objects a if there is a β-computation route from A to a modulo zero execution, then there is also a ν-computation route from A to a. Analogously in other cases. Thus, to show translatability, we want to prove $\forall A \forall a (A \twoheadrightarrow^0_\beta a \Longrightarrow A \twoheadrightarrow_\nu a)$ or $\forall A \forall a (A \twoheadrightarrow_\nu a \Longrightarrow A \twoheadrightarrow^0_\beta a)$ or both.

Hence, in order to demonstrate the mutual untranslatability, it is sufficient to find computations routes $\twoheadrightarrow^0_\beta$ and $\twoheadrightarrow_\nu$ (essentially, edges in the above schemes) that start in A but produce different results (even modulo zero execution), i.e., we want to find A and a that do not satisfy the above conditions.

Let us start with the $\twoheadrightarrow_\nu$ to $\twoheadrightarrow^0_\beta$ direction. Suppose that A is $[[\lambda x[\mathbf{Div}\ x\ \mathbf{0}]\ \mathbf{5}]$. Then $[[\lambda x[\mathbf{Div}\ x\ \mathbf{0}]\ \mathbf{5}] \twoheadrightarrow_\nu \varnothing$, but $[[\lambda x[\mathbf{Div}\ x\ \mathbf{0}]\ \mathbf{5}] \twoheadrightarrow^0_\beta [\mathbf{Div}\ \mathbf{5}\ \mathbf{0}]$. Clearly, we have varying results, because $\varnothing \neq [\mathbf{Div}\ \mathbf{5}\ \mathbf{0}]$. Since we can form a construction that violates our translatability conditions, we can conclude that it is not the case that semantic computation can be translated into syntactic computation for all constructions. Thus, $\forall A \forall a (A \twoheadrightarrow_\nu a \Longrightarrow A \twoheadrightarrow^0_\beta a)$ and, consequently, $\Pi(\nu,\beta)$ and $\Pi(full)$ do not hold.

Now, let us consider the other direction $\twoheadrightarrow^0_\beta$ to $\twoheadrightarrow_\nu$. Let us suppose that A is $[\mathbf{Succ}\ \mathbf{0}]$. Then $[\mathbf{Succ}\ \mathbf{0}] \twoheadrightarrow^0_\beta \cdots$, but at the same time $[\mathbf{Succ}\ \mathbf{0}] \twoheadrightarrow_\nu 1$. Clearly $\cdots \neq 1$, more precisely, $[\mathbf{Succ}\ \mathbf{0}] \neq 1$. Hence, once again, we have different results – we can find a construction whose semantic computation is untranslatable into syntactic one. Therefore, we can conclude that $\forall A \forall a (A \twoheadrightarrow^0_\beta a \Longrightarrow A \twoheadrightarrow_\nu a)$ and, consequently, $\Pi(\beta,\nu)$ do not hold either.

4 Interaction of syntactic and semantic computations

So far we have defined two notions of computation and specified their four possible combinations (i.e., basic computations types of constructions). The natural next question seems to be: are there any interactions between these four computation types we can trace? For example, if we compose together two constructions one of which is only semantically computable and the other only syntactically computable, what computation type will inherit the resulting construction?

Unfortunately, the answer is no, we cannot generally track the interaction of these two notions of computations. The main reason for this is that semantic computability often depends on individual valuations. Take e.g., [x **5** **0**], whether this construction is v-computable depends entirely on the valuation v, specifically, what function the construction x v-constructs. If it v-constructs e.g., addition Add, then it is v-computable, if division Div, then non-v-computable, etc. There is, however, a way around it: we can restrict our attention only to closed constructions, i.e., constructions with no free variables. This will eliminate the problem of [x **5** **0**] and other similar constructions being semantically computable/non-computable based on valuations alone.

Thus, we are now faced with the following narrower problem: can we systematically study the interaction between syntactically and semantically computable *closed* constructions? And by systematic study we mean being able to determine the computation type of the resulting construction out of the computation types of the parts it was composed of.[14] In other words, we take "systematic" to be synonymous with "predictable" in the sense that the same inputs should produce the same outputs.

For this purpose we will introduce the following metalanguage notation:

– $\boldsymbol{a}+\boldsymbol{b}=\boldsymbol{c}$ can be read as: a composition of two constructions of computation types $\boldsymbol{a}$ and $\boldsymbol{b}$ results in a construction of computation type $\boldsymbol{c}$, where $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$ are metavariables for ■■, ■□, □■, □□.

For example, the construction [**Succ** **0**] will be associated with the following statement of our new syntax: □■ + □■ = □■.

Our claim is that computation types of constructions alone cannot determine the computation type of the resulting construction composed of them. We can prove this by contradiction.

Reductio hypothesis: Computation types of constructions alone can determine the computation type of the resulting construction composed of them. From our Reduction hypothesis follows that no composition of two different computation types can ever lead to different results. What this entails is essentially that the results should be

[14] For simplicity, we consider only compositions composed of two constructions, one constructing a function, the other its argument.

always foreseeable, i.e., combinations of same computability-types of constructions should lead to the same outcomes.

Working assumption: Combination of two non-β-computable constructions should result in non-β-computable construction (i.e., $\square\blacksquare + \square\blacksquare = \square\blacksquare$ in our symbolic notation). For example, **Succ** is not β-computable (i.e., of type $\square\blacksquare$) and so is **0**. And if we compose them together, we get another non-β-computable construction [**Succ 0**] (also of type $\square\blacksquare$). So from the composition of two non-β-computable constructions we get another non-β-computable constructions. Hence, $\square\blacksquare + \square\blacksquare = \square\blacksquare$ holds, just as our working assumption predicted.

But now consider slightly different case. Again we have two constructions $[\lambda x\, x]$ and **1** that are also both non-β-computable (i.e., of types $\square\blacksquare$). However, if we compose them together, we get $[[\lambda x\, x]\, \mathbf{1}]$, which is now a β-computable construction (of type $\blacksquare\blacksquare$). Hence, $\square\blacksquare + \square\blacksquare = \blacksquare\blacksquare$. And thus, contradiction arises, since we can now derive that $\square\blacksquare = \blacksquare\blacksquare$. So we can have compositions of constructions of same computation types leading to different results. Hence, our working hypothesis is false. Therefore, we need more than just the type of computation of the corresponding construction to determine the computation type of the resulting construction.

5 Final remarks

In this paper we have examined two notions of computation native to TIL, a syntactic one and a semantic one, which roughly correspond to term rewriting and term interpretation from lambda calculus. We showed that these two notions are mutually untranslatable in TIL and that we can in general recognize four different computation types among TIL constructions based on the combination of their syntactic and semantic computability. Finally, we showed that the computation types of constructions alone cannot determine the computation type of the resulting construction composed of them.

6 Appendix

6.1 Constructions

The traditional definition of TIL constructions goes as follows (original formulation by Tichý (1988), we follow Duží et al. (2010) with minor deviations):[15]

Definition 5 (*Constructions*)

1. The *variable* x is a *construction* that constructs an object O of the respective type dependently on a valuation. We say that it v-constructs O.
2. Where X is any object, ${}^0X$ is the *construction trivialization*. It constructs X without any change.
3. The *composition* $[X_0 X_1 \ldots X_m]$ is the following *construction*. If X v-constructs a function f of type $(\alpha\beta_1 \ldots \beta_m)$ and $X_1 \ldots X_m$ v-construct objects $b_1, \ldots, b_m$ of types $\beta_1, \ldots, \beta_m$, respectively, then the *composition* $[X_0 X_1 \ldots X_m]$ v*-constructs* the value (an object of type α, if any) of f on the tuple-argument $\langle b_1, \ldots, b_m \rangle$. Otherwise, it is a v*-improper construction*, i.e., *construction* that does not construct anything.
4. The *closure* $[\lambda x_1 \ldots x_m Y]$ is the following *construction*. Let $x_1, \ldots x_m$ be pairwise distinct variables v-constructing objects of types $\beta_1, \ldots, \beta_m$ and Y a construction v-constructing an object of type α. Then $[\lambda x_1 \ldots x_m Y]$ is the *construction closure*. It v*-constructs* the following function f of type $(\alpha\beta_1 \ldots \beta_m)$: let $\langle b_1, \ldots, b_m \rangle$ be a tuple of objects of types $\beta_1 \ldots \beta_m$, respectively, and v' be a valuation that associates x_i with b_i and is identical to v otherwise. Then the value of function f on argument tuple $\langle b_1, \ldots, b_m \rangle$ is the object of type α v'-constructed by Y. If Y is v'-improper, then f is undefined on $\langle b_1, \ldots, b_m \rangle$.
5. The *single execution* ${}^1X$ is the *construction* that either v-constructs the object v-constructed by X or, if X v-constructs nothing, is v-improper.
6. The *double execution* ${}^2X$ is the following *construction*: let X be any object, the *double execution* ${}^2X$ is v*-improper* if X is a non-construction or if X does not v-construct a construction or if X v-constructs a v-improper construction. Otherwise, let X v-construct a construction X' and let X' v-construct and object X'', then ${}^2K$ v-constructs X''.
7. Nothing other is a *construction*, unless it follows from 1–6.

6.2 Ramified type theory

We follow the specification from Tichý (1988):

Definition 6 (*Ramified type theory of TIL*) Let B be a base, i.e., a set of atomic types.

[15] An alternative formulation can be found in Raclavský et al. (2015).

1. (t_1i) Every member of B is a *type of order* 1 *over* B.

($t_1$ii) If $0 < m$ and $\alpha, \beta_1 \dots \beta_m$ are *types of order* 1 *over B*, then the collection $(\alpha\beta_1 \dots \beta_m)$ of all m-ary (total and partial) mappings from $\beta_1 \dots \beta_m$ into α is also a *type of order* 1 *over* B.

($t_1$iii) Nothing is a *type of order* 1 *over* B unless it follows from (t_1i) and ($t_1$ii).

2.(c_ki) Let α be any *type of order* k *over* B. Every variable ranging over α is a *construction of order* k *over* B. If X is of (i.e., belongs to) type α, then ^0X, ^1X, and ^2X are *constructions of order* k *over* B. Every variable ranging over α is a *construction of order* k *over* B.

($c_k$ii) If $0 < m$ and $X_0, X_1, \dots, X_m$ are *constructions of order* k, then $[X_0\, X_1 \dots X_m]$ is a *construction of order* k *over* B. If $0 < m$, α is a *type of order* k *over* B, and Y as well as the distinct variables $x_1, \dots, x_m$ are *constructions of order* k *over* B, then $[\lambda_\alpha x_1 \dots x_m\ Y]$ is a *construction of order* k *over* B.

($c_k$iii) Nothing is a *construction of order* k *over* B unless it follows from (c_ki) and ($c_k$ii).

Let $*_k$ be the collection of *constructions of order k over* B. The collection of *types of order* $k+1$ *over* B is defined as follows:

(t_{k+1}i) $*_n$ and every *type of order n* is a *type of order* $k+1$.

(t_{k+1}ii) If $0 < m$ and $\alpha, \beta_1, \dots, \beta_m$ are *types of order* $k+1$ *over* B, then the collection $(\alpha\beta_1 \dots \beta_m)$ of all m-ary (total and partial) mappings from $\beta_1, \dots, \beta$ into α is also a *type of order* $k+1$ *over* B.

(t_{k+1}iii) Nothing is a *type of order* $k+1$ *over* B unless it follows from (t_{k+1}i) and (t_{k+1}ii).

6.3 Collisionless substitution

Originally defined by Duží et al. (2010):

Definition 7 (*Collisionless substitution*) Let x be a variable and C, D any kinds of construction. If x is not free in C, then *the result of substituting* D *for* x *in* C is C. Assume now that x is free in C. Then:

(a) If C is x, then *the result of substituting* D *for* x *in* C is D. If C is ^1X or ^2X, then *the result of substituting* D *for* x *in* C is ^1Y or ^2Y, where Y is the result of substituting D for x in X.

(b) If C is $[X\,X_1 \dots X_m]$, then *the result of substituting* D *for* x *in* C is $[Y\,Y_1 \dots Y_m]$, where $Y, Y_1, \dots, Y_m$ are the results of substituting D for x in $X, X_1, \dots, X_m$, respectively.

(c) Let C be of the form $[\lambda x_1 \dots x_m\, Y]$; for $1 \leqslant i \leqslant m$, let $y_i = x_i$ if x_i is not free in D, and otherwise the first variable ν-constructing entities of the same type as x_i, not occurring in C, not free in D, and distinct from $y_1, \dots, y_{i+1}$. Then *the result of substituting* D *for* x *in* C is $[\lambda y_1 \dots y_m\, Z]$, where Z is the result of substituting D for x in the result of substituting y_i for $x_i (1 \leqslant i \leqslant m)$ in Y.

To simplify the notation, let $C, D_1, \dots, D_n$ be arbitrary constructions, $x_1, \dots, x_n$ variables. Then, for $1 \leqslant i \leqslant n$, $C(D_i/x_i)$ will represent the result of substituting D_i for x_i in C.

6.4 β-reduction

We follow the specification of β-reduction from Duží (2017):[16]

Definition 8 (β-*reduction*) Let C be a construction typed to v-construct object of type α, x_1 and y_1 variables typed to v-construct objects of type $\beta_n, \dots, x_n$ and y_n variables typed to v-construct objects of type β_n and $[\lambda x_1 \dots x_n\, C]$ is typed to v-construct object of type $(\alpha\beta_1 \dots \beta_n)$, then

$$[[\lambda x_1 \dots x_n\, C]\, y_1 \dots y_n] \longrightarrow_\beta C(y_1/x_1 \dots y_n/x_n)$$

where $C(y_1/x_1 \dots y_n/x_n)$ is the construction that arises from C by collisionless substitution of y_1 for all the occurrences of $x_1, \dots, y_n$ for all the occurrences of x_n, C v-constructs object of type α, x_1 and y_1 v-construct objects of type $\beta_1, \dots, x_n$ and y_n v-construct objects of type β_n, and $[\lambda x_1 \dots x_n\, C]$ v-constructs a function of type $(\alpha\beta_1 \dots \beta_n)$.

6.5 n-execution

In this section, we introduce the construction of multiple execution we will call n-execution (for $n \geqslant 0$), which is a generalization of TIL's native constructions execution and double execution (see Section 6.1 above). Further, we will show that trivialization can be understood as its limiting case when $n = 0$. Our generalization is based on the following three simple observations that keep reappearing in TIL literature from time to time:

Observation 1. If we can have single execution and double execution, then, in theory, there should be nothing preventing us from introducing even triple execution, quadruple execution and so on. And if that is the case, then we can generalize this observation and introduce the notion of n-execution where n is the number of consecutive executions.[17]

Observation 2. If we should understand ${}^1X$ as 'execute X' (i.e., '^{1}' = one execution) and ${}^2X$ as 'execute X and then execute its result X'' (i.e., '^{2}' = two executions), then, arguably the most natural reading of ${}^0X$—if we have never heard of trivialization—is 'do not execute X' (i.e., '^{0}' = zero executions).[18]

[16] We utilize here so called restricted β-reduction by name (see Duží 2017), but other variants can be employed as well.

[17] See e.g., Materna (1998), Duží et al. (2010), Raclavský et al. (2015).

[18] See e.g., Raclavský (2003), Jespersen (2017).

Observation 3. Multiple execution for $n > 2$ is reducible into iterated double execution.[19]

Definition 9 (n-*execution*) For any object X we shall speak of the n-*execution* of X and symbolize it as ^nX, where $n \geqslant 0$.

1. If $n = 0$, then ^0X is X (where X is either a construction or a non-construction). More specifically, to carry out ^0X, we start with X and do nothing further with it, not even valuation. It is never v-improper.
2. If $n \geqslant 1$, there is a need to distinguish two main cases (the second has three further subcases):
   (a) X is a non-construction: then ^nX is v-improper construction.
   (b) X is a construction:
      i. $n = 1$: then ^nX v-constructs the same object as does X (if any).
      ii. $n = 2$: then ^nX v-constructs the object v-constructed by X' (assuming ^nX v-constructs X'). Otherwise, it is v-improper.
      iii. $n > 2$: then ^nX v-constructs the same object (if any) as does ^sX where s is a sequence of $n-1$ iterated n-executions with $n = 2$. For example, ^3X can be reduced into $^2(^2X)$.

From the type-theoretical perspective, if X is of (i.e., belongs to) some type α of order k over B, then ^nX (where $n \geqslant 0$) is a *construction of order* k *over* B.[20]

To get a better feeling of how n-execution works for cases with $n > 1$, consider the following cases:

1. $n = 2$: same as Tichý's original definition above.
2. $n = 3$: analogously to the case above, but the construction X' v-constructed by X has to be v-constructing another (proper) construction X'', otherwise it is v-improper.
3. $n = 4$: analogously to the case above, but X'' has to be v-constructing yet another (proper) construction X''', otherwise it is v-improper.

   $\ldots$ etc.

Specific instances of n-execution (for $n = 0, 1, 2, \ldots$) will be called 0-*execution*, 1-*execution*, 2-*execution*, etc. For example, ^{05} is an example of 0-execution, ^1x is an example of 1-execution, ^{25} is an example of 2-execution, etc.

To conclude, on this approach, trivialization, execution, and double execution are just specific instances of n-execution for some $n \geqslant 0$, with trivialization being an

[19] Originally noted by Petr Kuchyňka, see e.g., Raclavský (2003), Raclavský et al. (2015).

[20] For more about ramified type hierarchy, see Section 6.2.

alternative name for the degenerate case when $n = 0$ (informally: 'do not execute', 'do nothing' or—as Tichý put it—'leave it as it is' (Tichý (1988), p. 63).

Remark 7 The case of $n = 0$ is a special one also from another point of view – the appearance of '^{0}', i.e., 0-execution, binds free variables in contrast to 1-execution, 2-execution, etc. (see e.g., Duží et al. (2010), p. 47). This is, after all, to be expected. As we mentioned, 0-execution ^0X is essentially an instruction to 'do nothing with X' and by binding the free variables we are making sure that nothing is indeed done with X, not even valuation or substitution.

References

Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2013. ISBN 9780444854902. URL https://books.google.cz/books?id=ZdydJZVue50C.

Marie Duží. If structured propositions are logical procedures then how are procedures individuated? *Synthese*, 0(0):0, 2017. ISSN 0039-7857. doi: https://doi.org/10.1007/s11229-017-1595-5. URL https://doi.org/10.1007/s11229-017-1595-5.

Marie Duží, Bjørn Jespersen, and Pavel Materna. *Procedural Semantics for Hyperintensional Logic: Foundations and Applications of Transparent Intensional Logic*. Logic, Epistemology, and the Unity of Science. Springer, Dordrecht, 2010. ISBN 9789048188123. doi: 10.1007/978-90-481-8812-3. URL http://link.springer.com/10.1007/978-90-481-8812-3.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989. ISBN 9780521371810.

Bjørn Jespersen. Anatomy of a Proposition. *Synthese*, pages 1–40, 2017. ISSN 1573-0964. doi: 10.1007/s11229-017-1512-y. URL https://doi.org/10.1007/s11229-017-1512-y.

Per Martin-Löf. *Intuitionistic type theory*. Studies in proof theory. Bibliopolis, 1984.

Pavel Materna. *Concepts and Objects*. Acta Philosophica Fennica. Helsinki: Philosophical Society of Finland, Vol. 63. 1998.

Pavel Materna. Equivalence of Problems (An Attempt at an Explication of Problem). *Axiomathes*, 23(4):617–631, dec 2013. ISSN 1122-1151. doi: 10.1007/s10516-012-9201-4. URL http://link.springer.com/10.1007/s10516-012-9201-4.

Yiannis Nicholas Moschovakis. A Logical Calculus of Meaning and Synonymy. *Linguistics and Philosophy*, 29(1):27–89, 2006. doi: 10.1007/s10988-005-6920-7.

Reinhard Muskens. Sense and the computation of reference. *Linguistics and Philosophy*, 28(4):473–504, 2005. doi: 10.1007/s10988-004-7684-1.

Giuseppe Primiero and Bjørn Jespersen. Two Kinds of Procedural Semantics for Privative Modification. *Lecture Notes in Artificial Intelligence*, 6284:251–271, 2010.

Jiří Raclavský. Executions vs. Constructions. *Logica et Methodologica*, 7:63–72, 2003.

Jiří Raclavský, Petr Kuchyňka, and Ivo Pezlar. *Transparentní intenzionální logika jako characteristica universalis a calculus ratiocinator*. Brno: Masaryk University (Munipress), Brno, 2015.

A. Sierszulska. On Tichy's determiners and Zalta's abstract objects. *Axiomathes*, 16 (4):486–498, jan 2007. ISSN 1572-8390. doi: 10.1007/s10516-006-0002-5. URL http://link.springer.com/10.1007/s10516-006-0002-5.

Pavel Tichý. *The Foundations of Frege's Logic*. Foundations of Communication. de Gruyter, 1988. ISBN 9783110116687.

Jean van Heijenoort. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*. Source Books in the History of the Sciences. Harvard University Press, 1977. ISBN 9780674324497.