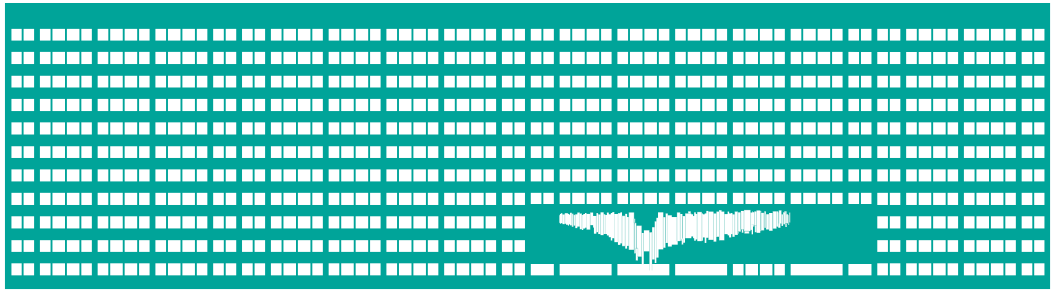
VŠB TECHNICKÁ UNIVERZITA OSTRAVA

VSB TECHNICAL UNIVERSITY OF OSTRAVA



Syntactic vs semantic consistency of a hyperintensional system with procedural semantics

Marie Duží, Bjørn Jespersen

Czech Republic

Procedures in logic and programming languages

- ***Transparent Intensional Logic*** (TIL): hyperintensional, typed λ-calculus with ***procedural*** semantics
  - ***Trivialization*** and ***Double Execution***: a pair of twin procedures
- Trivialisation, when applied to another procedure *C* (^0C), makes *C* displayed as an argument to operate on; *C* occurs hyperintensionally
- Double Execution, when applied to another procedure *C* (^2C), produces the product of what is produced by *C*;
- Hence, by applying the substitution method, we can *first modify C,* and *second execute* it: $^2[^0Sub\ A\ ^0x\ ^0C(x)]$
- higher-order functional programming languages
  - *Eval* → makes it possible to *modify* the procedure encoded by a piece of quoted string
  - *Apply* → the actual call of the procedure (*execution*)

Substitution, β-reduction 'by value'

$$[\lambda x\ F(x)\ A] = {}^{2}[{}^{0}Sub\ [{}^{0}Tr\ A]\ {}^{0}x\ {}^{0}F(x)]$$

1. *Eval F* *(modify F by supplying an argument a for* x*)*

   a) *A: execute A in order to obtain the value a;*
   if A is *v-improper, then the whole Composition is v-improper (stop); else*:

   b) $[{}^{0}Tr\ A]$: *obtain Trivialization of ("pointer at")* the argument *a* produced by *A*

   c) $[{}^{0}Sub\ [{}^{0}Tr\ A]\ {}^{0}x\ {}^{0}F]$: *substitute this 'pointer at a' for x into 'the body' F;*
   the resulting procedure *F'* is properly parametrised

2. *Apply (execute the modified F)*: $^{2}[{}^{0}Sub\ [{}^{0}Tr\ A]\ {}^{0}x\ {}^{0}F]$

- in Computational λ-Calculus (***CLC***), the modification and execution of computations are expressed using *monadic bind*, typically written:

$$x \leftarrow A;\ F(x)$$

- *Eval F*: run *A*, get a result, plug the result into *F.*
- *Apply*: run the adjusted *F*.

Problems arising when dealing with procedures

- Trivialisation ${}^0C$ *displays* C, C occurs in ***displayed mode***, within a ***hyperintensional*** context.
  - Any sub-procedures occurring within ${}^0C$ also occur displayed, i.e. the product (if any) of C is irrelevant. In particular, all the variables occurring in ${}^0C$ are ***0-bound***, which is ***dominant over λ-binding***
- ***Double Execution*** ${}^2C$ goes beyond C, as ${}^2C$ produces what is produced by C. This 'product-hopping' takes ${}^2C$ beyond its complement, C, which is what upsets **variable-binding**, and which is the **central problem** of this paper.
- While ${}^0C$ *displays* procedure C as an operand or functional argument, ${}^2C$ *cancels* the displaying effect: ***${}^{20}C = C$***
  - There are cases where due to Double Execution, ***0-bound*** occurrences of variables ***become free***, or ***new free*** occurrences of variables crop up;
    it undermines the definition of correct substitution, jeopardises the validity of the rules of λ-conversion, in particular β-conversion.

Content

- Foundations of TIL 2010
- Demonstration of inconsistencies by examples
- Proposal of the solution: towards TIL 2025 (syntactic consistency)
  - Syntactic vs. semantic computation
- Comparison with other higher-order logical systems that explicate structured hyperintensions as computations
  - HTLC ≈ PLLe ≈ CLCe
- Compensation Principle; proof of consistency
- Kosterec's evaluation strategy and semantic consistency
- Concluding remarks

Foundations of TIL

- Intensional λ-calculi:

- Hyperintensional λ-calculi:

- Abstract, algorithmically structured ***procedure*** – Church's ***function-in-intension***
- TIL is a hyperintensional λ-calculus of *partial functions*; the meaning procedure can fail to produce anything by being *v*-improper
- The product can be a 1st-order object (nullary function) like individual, number, or a mapping, or a higher-order object, i.e. a mapping involving procedures

Foundations of TIL; type hierarchy

- The stratified TIL ontology is organized into ramified hierarchy of types

Types of order 1: non-procedural entities

- Base *B* = {ο, ι, τ, ω} – atomic *types of order 1*
- Where α, β_1,…,β_n are types of order 1, the set of partial mappings $(\beta_1 \times \ldots \times \beta_n) \rightarrow \alpha$, denoted '$(\alpha\ \beta_1 \ldots \beta_n)$', is a functional *type of order 1*.

Types of order n>1: mappings involving procedures

- Procedures producing 1st-order objects belong to a ***type of order 2***, denoted '$*_1$'.
  - Mappings $(\alpha\ \beta_1 \ldots \beta_n)$, involving $*_1$ in their domain or range are *types of order 2*;
- Procedures producing entities of order 1 or 2, and partial functions involving such procedures, belong to a type $*_2$, a ***type of order 3***;
- Procedures belonging to a type $*n$, which produce entities of order $n \geq 1$, and partial functions involving such procedures, belong to a ***type of order*** $n + 1$;
- and so on *ad infinitum*.

Foundations of TIL; procedures

- Entities on which procedures operate are *not* their constituents; they must be supplied or displayed by ***atomic procedures***:

- ***Trivialisation*** ^0a (similar role as a constant in λ-calculi)
- ***Variables*** *x, y, p, q, c*, … produce entities dependently on a valuation; they *v*-produce
  - ***Molecular procedures***:
- ***Composition*** $[X\ Y_1 \ldots Y_m]$ is the procedure of ***applying*** a function;

  $X \rightarrow (\alpha\beta_1 \ldots \beta_m)$; $Y_1 \rightarrow \beta_1$, …, $Y_n \rightarrow \beta_n$;

  $[X\ Y_1 \ldots Y_m] \rightarrow \alpha$
- ***Closure*** $[\lambda x_1 \ldots x_m\ Y]$ is the procedure of producing a function;

  $x_1 \rightarrow \beta_1$, …, $x_n \rightarrow \beta_n$; $Y \rightarrow \alpha$

  $[\lambda x_1 \ldots x_m\ Y] \rightarrow (\alpha\beta_1 \ldots \beta_m)$
- *Execution* ^1X ; ***Double Execution*** ^2X

Two modes of procedure's occurrence

- ***Displayed mode***: any sub-procedure (including variables) of ${}^0C$ occurs in displayed mode;
  - ***variables*** occurring in *C* are **0-bound;** **${}^0[\lambda x\,[{}^0+\,x\,{}^01]]$, *x* is 0-bound**
  - A displayed procedure *C* can become the argument of a function, and it is possible to operate on *C* itself.
- ***Executed mode***: if a procedure *C* does not occur displayed, it occurs in executed mode by ***default***;
  - ***variables*** can occur in *C free* or *λ-bound*; **$[\lambda x\,[{}^0+\,x\,{}^01]]$, *x* is λ-bound**
- ***Variables can occur free or λ-bound only in the executed mode; otherwise, they are bound by Trivialisation, i.e. 0-bound***
- The rule of ***^{20}-elimination*** is valid for every procedure *C*: ${}^{20}C = C$

From TIL 1988 to TIL 2010

- **Tichý** (1988, §17) defined correct substitution in a restrictive manner. He was careful not to substitute into a context within the scope of a Trivialization, i.e., a hyperintensional context.
  - In TIL, whether λ-bound or not, as soon as a variable occurs within the scope of Trivialization, it is 0-bound, hence displayed, and thus not free for substitution.
- Tichý proved the basic theorem covering the validity of substitution (and thus also the validity of the λ-conversions), namely the ***Compensation Principle***, thereby proving the *consistency* of the results of substitution.
- Yet, proving this principle ***only for procedures of order 1***, Tichý did not logically operate within hyperintensional contexts, and so he did not get around to considering the fact that Double Execution cancels the effect of Trivialization.

From TIL 1988 to TIL 2010

- ***Duží, Jespersen & Materna*** (2010) aimed to extend the definitions of a free occurrence of a variable and correct substitution to also legislate the free/bound status of those occurrences that, under *specific circumstances*, become free due to the interaction between Double Execution and Trivialization. However, we went ***too far*** …
- Since (2010), the definition of substitution, as well as the proof of its correctness, appeared to be a completed task. Over the years, TIL has been applied not least to natural-language processing, and when analysing natural-language sentences, everything checked out.
- However, ***Kosterec*** (2020), (2021) demonstrated inconsistencies in TIL (2010). These (extreme cases) are due to *too liberal a definition of variables occurring free* for substitution, which in turn goes hand in hand with too liberal a definition *of the execution mode* for occurrences of procedures.

To ‘clear the slum and be done with it’?

- To eliminate the causes of the problem in any of these ways:

- *Drop the transitivity of execution! Why include a procedure that is causing you trouble? **Double Execution** is a nice-to-have, not a need-to-have.*
  - *No way;* we need Double Execution when operating on hyperintensional contexts. Recall, first, modify the displayed procedure, and second, execute it
- *Drop **Trivialization** as a procedure! Procedures always occur in executed mode, because presenting entities is what procedures are all about. So, there is no such thing as a procedure occurring non-executed.*
  - Not quite right. Dropping Trivialization would cause us to lose the ability to tap into the hyperintensional features of our logic. Procedures are themselves objects sui generis. These are the complements that an agent is related to in the case of hyperintensional attitudes

To ‘clear the slum and be done with it’?

- *Build up a **parallel system**, in which the default is that procedures occur in displayed, rather than executed, mode. Then Trivialization is no longer required to obtain the effect of displayed procedures; this you get for free. Instead, what is needed is a procedure to take an input from occurring in displayed to occurring in executed mode, and you already have that, namely, (Single) Execution.*
  - This is feasible, of course. But doing so would be pointless. Also in the parallel system, one would have to solve the same problems as demonstrated by Duží (2025).
- Hence, the problem must indeed be addressed head-on.

the heart of the problem

- mixing together the ***syntactic structure*** of a procedure with its ***semantic computation***.
- Pezlar (2019) helpfully distinguishes between ***syntactic*** computation, i.e. ***λ-conversions***, and ***semantic*** computation, which is the *evaluation* of the product a procedure produces with respect to a ***valuation v***.
- To obtain ***syntactic consistency***,

it must be decisive at the syntactic structural level, which variables are free for substitution, prior to any computation is launched.

TIL 2010 critical flawed definitions

Closure is the procedure of producing a function by abstraction of the values of λ-bound variables:

- [$\lambda x_1 \ldots x_m$ *Y*] is the *procedure* λ-*Closure* (or *Closure*). It *v-produces* the function $f/(\alpha\beta_1 \ldots \beta_m)$ such that the value of *f* at $\langle B_1,\ldots,B_m\rangle$, if any, is the α-entity *v*($B_1/x_1,\ldots,B_m/x_m$)-produced by *Y*.
- ***Problem***: some of the variables x_i can crop us as "free" when evaluating *Y,* due to Double Execution.

Example $[[\lambda x\ ^2y]\ z]$

- Assume that $y \Rightarrow_v x \Rightarrow_v k$ and $z \Rightarrow_v j$, where $k \neq j$.

$$[[\lambda x\ ^2y]\ z] \Rightarrow_\beta {}^2y \Rightarrow_v k$$ (variable x does not occur in ^2y)

- Definition of ***Closure***: $[[\lambda x\ ^2y]\ z] \Rightarrow_v j$
  - the value of the function produced by the Closure $[\lambda x\ ^2y]$ at the argument j (v-produced by z) is the object ***v*(*j*/*x*)-produced by** ^2y. Since y ***v*-produces** x, which in turn ***v*(*j*/*x*)-produces** j, the result of functional application is the value j.
- *in*validity of this restricted β-reduction by name. (???)
- *Variable x v-produced by y cropped up as free during the semantic computation of* ^2y*, which is not correct. We cannot substitute j for x, as x would become λ-bound;* ***conflict of variables***

Example. $[\lambda y\ [^2x\ ^0C]\ ^0D]$

Definition 1.4 (*free variable, bound variable*)

- If an occurrence of ξ is ^{0}*bound* in a constituent ^0D of ^2X and this occurrence of D is a constituent of X' v-produced by X, then if the occurrence of ξ is free, λ-bound in D, it is *free, λ-bound in C.*
- Let $x \Rightarrow_v y \Rightarrow_v ID$: the identity function; $^2x \Rightarrow_v ID$; $C \neq D$.

$$[\lambda y\ [^2x\ ^0C]\ ^0D] \Rightarrow_\beta [^2x\ ^0C] \Rightarrow_v C$$

- However, $\lambda y\ [^2x\ ^0C]$ v-produces the function that assigns to the argument D the value that is $v(D/y)$-produced by $[^2x\ ^0C]$.
- Since $x \Rightarrow_v y$ and $y \Rightarrow_{v(D/y)} D$, the result would be D

TIL 2010 critical flawed definitions

- **Definition 2.15 (*Displayed vs executed procedure*)** Let *C* be a procedure and *D* a subprocedure of *C.*

v. If *C* is identical with ^{2}X and *D* is identical with *X*, **or ^{0}D occurs as a constituent of *X* and this occurrence of *D* occurs as a constituent of *Y* *v*-produced by *X***, then the occurrence of *D* is *executed* in *C*.

- Hence, variables occurring in *D become* free; which they shouldn't.

Example; If-then-else function

- 'If *P* then *C*, else *D*' is defined as follows.

$$[^0\mathit{If\_then\_else}\ P\ {}^0C\ {}^0D] =_{df} {}^2[{}^0\mathrm{I}\ \lambda c\ [P \wedge c={}^0C] \vee [\neg P \wedge c={}^0D]]$$

  - *Types*: $P \to o$; $C,D \to \alpha$; $c \to *_n$; ${}^2c \to \alpha$; $\mathrm{I}/(*_n\,(o*_n))$: the singularizer

1) The ***choice*** between *C* and *D*: $[{}^0\mathrm{I}\ \lambda c\ [P \wedge c={}^0C] \vee [\neg P \wedge c={}^0D]]$
2) the ***selected procedure is executed***; hence, Double Execution.

The decision as to which of the two procedures is to be executed depends on the value ***v-produced*** by *P.*

Hence, from the ***syntactic/structural*** point of view, both procedures are ***displayed***, and *none* of the variables occurring in either *C* or *D* is free for substitution.

But, as one of them is *v*-produced as a constituent of the result, it would occur in the ***executed*** mode. This is not correct.

Leibniz's rules of substitution of identicals

The rule of substitution in an extensional context.

- ***v-congruent*** procedures can be substituted

$$\frac{D \approx_v D'}{C \approx_v C(D'/D)}$$

The rule of substitution in an intensional context.

- ***Equivalent*** *procedures can be substituted*

$$\frac{D \approx D'}{C \approx C(D'/D)}$$

The rule of substitution in a hyper-intensional context

- ***Isomorphic*** *procedures can be substituted*

$$\frac{D \cong D'}{C \cong C(D'/D)}$$

a counterexample (?) to the rule of *substitution of v-congruent procedures occurring extensionally*.

- Let $x \rightarrow_v 1$; $z \rightarrow_v 3$; $y \rightarrow_v [^0+ x\ ^0 2]$; hence, $^2y \rightarrow_v 3$. Let us compute:

$$[\lambda x\ [^0\neq x\ z]\ ^0 3]$$

$$^2y \approx_v z;\ y \approx_v\ ^0[^0+ x\ ^0 2].$$

$[\lambda x\ [^0\neq x\ z]\ ^0 3] \Rightarrow_\beta [^0\neq\ ^0 3\ z] =_v\ ^0 F$

$[\lambda x\ [^0\neq x\ z]\ ^0 3] =_{(Subst)} [\lambda x\ [^0\neq x\ ^2 y]\ ^0 3] \Rightarrow_\beta [^0\neq\ ^0 3\ ^2 y] =_v\ ^0 F$

(substitution of ^2y for z)

$[\lambda x\ [^0\neq x\ ^2 y]\ ^0 3] =_{(Subst)} [\lambda x\ [^0\neq x\ ^{20}[^0+ x\ ^0 2]]\ ^0 3]$ ***conflict of variables***

(substitution of $^0[^0+ x\ ^0 2]$ for y)

$[\lambda x\ [^0\neq x\ ^{20}[^0+ x\ ^0 2]]\ ^0 3] = [\lambda x\ [^0\neq x\ [^0+ x\ ^0 2]]\ ^0 3] \Rightarrow_\beta [^0\neq\ ^0 3\ [^0+\ ^0 3\ ^0 2]] =_v\ ^0 T$

$[\lambda x'\ [^0\neq x'\ ^{20}[^0+ x\ ^0 2]]\ ^0 3] = [\lambda x'\ [^0\neq x'\ [^0+ x\ ^0 2]]\ ^0 3] \Rightarrow_\beta [^0\neq\ ^0 3\ [^0+ x\ ^0 2]] =_v\ ^0 F$

logical systems that explicate structured hyperintensions as computations

- **Tichý**: "The notion of [effective] construction [i.e. procedure] is correlative with a particular algorithmic ***computation***." (Tichý 1986, 526).
- **Moschovakis** (2006) argues that the meaning of a term *A* can be faithfully modelled by its 'referential intension' int(*A*), i.e. an abstract, idealized, not necessarily implementable ***algorithm which computes the denotation of A***.
- other higher-order logical systems, for instance, the ***Computational Lambda Calculus*** **(CLC)** of Moggi (1991), the ***Propositional Lax Logic*** **(PLL)** of Fairlough & Mender (1997), the ***Hyperintensional Typed Lambda Calculus*** **(HTLC)** of Fait and Primiero (2021).

higher-order systems which operate with procedures

- *Benton* et al. (1998) observed that there is an equivalence (via the Curry-Howard isomorphism) between CLC and PLL.
- Moreover, **Pezlar (2025)** shows that HTLC (inspired by TIL) equipped with a proper elimination rule for the hyperintensional operator * is equivalent to both CLC and PLL enriched by a simple axiom (E): $^{\circ}A \rightarrow A$, where $^{\circ}$ is a modality of the systems.
- Thus, we have the two systems PLLe and CLCe (PLL and CLC enriched with the axiom (E)) in the following relation to HTLC:

$$\text{HTLC} \approx \text{PLLe} \approx \text{CLCe}$$

- In addition, Duží (2025) shows that HTLC can be viewed as a subsystem of ***Dual TIL***, i.e. the system based on TIL in which the default mode in which procedures occur is displayed (as objects of predication) rather than executed (to deliver a value).

higher-order systems which operate with procedures

Pezlar (2025) makes an interesting observation:

- […] Arguably, [two] of the 'strangest' notions introduced in TIL are [the procedures known as] ***Trivialization*** and ***Double Execution*** […].
- However, the link between HTLC, PLL and CLC suggests that these notions might not be as singular as they previously seemed.
- For example, as we have shown, they are quite similar to the ***triv constructor*** and ***exe selector***, respectively, from HTLC, and thus also to the corresponding constructor and selector from PLL and CLC.
- In short, the discovered connection allows us to view ***Triv****ialization and Double* ***Exe****cution […] of TIL in a new light and for the first time connect Moggi's and Tichý's approaches to the notion of computation*.
- ***Monadic or staged systems like CLC must handle eval/apply carefully to preserve the scope and avoid conflict of variables. Evidently, if the tenet we adhere to in this paper (that it must be determinate and known at the syntactic level prior to any computation is launched which variables occur free for substitution) is not followed, bad staging might lead to variable capture or leakage.***

Solving inconsistencies; *towards TIL 2025*

Definition 1 (*Closure*)

- The *Closure* $[\lambda x_1 \ldots x_m\ Y]$ is the following *procedure*.
- Let $x_1, x_2, \ldots, x_m$ be pairwise distinct variables *v*-producing entities $B_1/\beta_1, \ldots, B_m/\beta_m$, Y a procedure *v*-producing an α-entity.
- Then $[\lambda x_1 \ldots x_m\ Y]$ *v-produces* the following function $f/(\alpha\beta_1 \ldots \beta_m)$.
- **Let $\{y_1, \ldots, y_k\} \subseteq \{x_1, \ldots, x_m\}$, $k \leq m$, be a subset of those variables that occur free in *Y* and let $v(B_1/y_1,\ldots,B_k/y_k)$ be a valuation identical with *v* at least up to assigning objects $B_1/\beta_1, \ldots, B_k/\beta_k$ to the free occurrences of variables $y_1, \ldots, y_k$, respectively.**
- **If *Y* is $v(B_1/y_1,\ldots,B_k/y_k)$-improper, then *f* is undefined on $\langle B_1,\ldots,B_m\rangle$. Otherwise the value of *f* on $\langle B_1,\ldots,B_m\rangle$ is the α-entity $v(B_1/y_1,\ldots,B_k/y_k)$-produced by *Y*.**

Solution (TIL 2025); $[\lambda y\ [^2x\ ^0C]\ ^0D]$

- According to the (new) definition of Closure, $\lambda y\ [^2x\ ^0C]$ *v*-produces the function that assigns to the argument *D* the value that is ***v(D/y)***-produced by $[^2x\ ^0C]$.
- Since *y* does not occur free in $\lambda y\ [^2x\ ^0C]$, the value of the *v*-produced function at the argument *D* is the product of $[^2x\ ^0C]$, i.e. ***C***.
- **The fact that $x \Rightarrow_v y$ and $y \Rightarrow_{v(D/y)} D$ is irrelevant** here, as in the Composition $[^2x\ ^0C]$ the variable *y* does not occur;
- hence, the Composition $[\lambda y\ [^2x\ ^0C]\ ^0D]$ does *not v(D/y)*-produce *D*. Everything checks out.

β-reduction; $[[\lambda x\,[[\lambda y^{2} y]\,{}^{0}x]]\,z]$

- the invalidity of the Church-Rosser theorem (???):

  (a) outer first: $[[\lambda x\,[[\lambda y\,{}^{2}y]\,{}^{0}x]]\,z] \Rightarrow_{\beta} [[\lambda y\,{}^{2}y]\,{}^{0}x] \Rightarrow_{\beta} {}^{20}x = \boldsymbol{x}$

  (b) inner first: $[[\lambda x\,[[\lambda y\,{}^{2}y]\,{}^{0}x]]\,z] \Rightarrow_{\beta} [[\lambda x\,{}^{20}x]\,z] = [[\lambda x\,x]\,z] \Rightarrow_{\beta} \boldsymbol{z}$

- The problem here is a bit more involved. We applied only syntactic computation ($\Rightarrow_{\beta}$) and then the universally valid rule of ^{20}-elimination, namely $^{20}C = C$.
- In the redux, *x* is o-bound; hence, it does not occur as free.
- But in (b), the variable *x* becomes free by substituting ^{0}x for *y* which is then Double-Executed. *But it should not become free, as it has not been free in the redux procedure*.

β-reduction; $[[\lambda x\,[[\lambda y\,^2y]\,^0x]]\,z]$

Apply α-conversion to rename the λ-bound variables.

- $[[\lambda x\,[[\lambda y\,^2y]\,^0x]]\,z] \rightarrow_\alpha [\lambda x'\,[[\lambda y\,^2y]\,^0x]\,z]$

Now everything is all right:

(a) outer first $\Rightarrow_\beta [[\lambda y\,^2y]\,^0x] \Rightarrow_\beta {}^{20}x = \boldsymbol{x}$

(b') inner first $\Rightarrow_\beta [[\lambda x'\,{}^{20}x]\,z] \Rightarrow_\beta {}^{20}x = \boldsymbol{x}$

- But the procedures $[[\lambda x\,[[\lambda y\,^2y]\,^0x]]\,z]$ and $[\lambda x'\,[[\lambda y\,^2y]\,^0x]\,z]$ are ***procedurally isomorphic***, and it is untenable that procedurally isomorphic procedures would produce different results.
- Our tenet: *If a variable does not occur free in the original redux procedure, it must not become free during the process of computation.*

β-conversion

Definition (β-*conversion*) Let C_i, $1 \leq i \leq n$, be procedures of the form $[[\lambda x_i\ Y_i]\ D_i] \rightarrow (\alpha_i \beta_i)$. The sequence of β-reductions $C_1 \Rightarrow_\beta C_2 \Rightarrow_\beta \ldots \Rightarrow_\beta C_n$ is *β-conversion.*

Claim. Provided in each step β-reduction (***by value***) is applied in such a way that we substitute only for those variables that occur **free in the body Y_1** of the original redux procedure C_1, the resulting β-conversion is strictly equivalent in the sense that for each valuation *v*, C_1 *v*-produces the same object as C_n or are both *v*-improper. This is valid for any permutation of the set $\{i \mid 1<i\leq n\}$. Hence, Church-Rosser theorem is valid. *Proof* is obvious.

(b) inner first: $[[\lambda x\ [[\lambda y\ {}^2y]\ {}^0x]]\ z] \Rightarrow_\beta [[\lambda x\ {}^{20}x]\ z] = [[\lambda x\ x]\ z] \Rightarrow_\beta \boldsymbol{z}$

- The mistake appears in the last reduction, where *z* is substituted for *x* though *x* is not free in the original redux.

β-reductions

a) *By name.* $[[\lambda x\ Y]\ D] \Rightarrow_{\beta n} Y(D/x)$;

Y(D/x) is the result of the correct simultaneous substitution of the *procedure D* for all the free occurrences of *x* in *Y*.

Not an equivalent transformation, loss of analytic information

b) *Restricted reduction by name.* $[[\lambda x\ Y]\ y] \Rightarrow_{\beta r} Y(y/x)$

c) *By value.* $[[\lambda x\ Y]\ D] \Rightarrow_{\beta v} Y(D/x)$;

Y(D/x) is the result of the correct simultaneous substitution of the *product of the procedure D* for all the free occurrences of *x* in *Y*. *substitution method* → bingo, no problems !!! ☺

β-reduction: $[\lambda x\, C(x)\, A] \mid\!\!- C(A/x)$

- In programming languages the difference between ‘by value’ and ‘by name’ revolves around the programmer’s choice of *evaluation strategy*.
  - Algol’60: “call-by-value” and “call-by-name”
  - Java: manipulates objects “by name”, however, procedures are called “by-value”
  - Clean and Haskell: “call-by-name”
- Similar work has been done since the early 1970s; for instance, Plotkin (1975) proved that the two strategies are ***not operationally equivalent***.
- *Chang & Felleisen (2012) call-by-need reduction by value.* But their work is couched in an untyped λ-calculus.

Towards TIL 2025

- **Definition 2 (*^{20}-elimination*, *^{20}-redux*, *^{20}-normal form*)** Let *C, X* be procedures and let ^{20}X be a sub-procedure of *C.* Then the replacement of an occurrence of ^{20}X by the procedure *X*, the result of which is procedure *C'*, is ^{20}-*elimination*: $C \rightarrow_{20} C'$. If a procedure *C* can be transformed into a procedure *D* by a finite number *n* ($n \geq 0$) of ^{20}-eliminations, and if no other ^{20}-redux occurs in *D*, then *D* is a ***^{20}-normal form*** of *C*.

  - In what follows, I consider only procedures in their ^{20}-normal form. When applying TIL to natural-language analysis, the resulting procedures are in this form. This is also the reason why the inconsistencies of TIL 2010 were not discovered earlier.

Towards TIL 2025

Definition (*Occurrence of a procedure in displayed vs. executed mode*)

Let *C* and *D* be procedures such that *D* is a sub-procedure of *C*.

i. If *C* is identical to *D*, then the *occurrence of D in C* is in ***executed*** mode.

ii. If *C* is identical to ${}^0X$, and *D* is a sub-procedure of *X*, then the *occurrence of D in C* is in the ***displayed*** mode.

iii. If *C* is identical to $[X_1\, X_2 \ldots X_m]$ and *D* is a sub-procedure of X_i, $1 \leq i \leq m$, then the *occurrence of D in C* is the same as the occurrence of *D* in X_i.

iv. If *C* is identical to $[\lambda x_1 \ldots x_m\, X]$ and *D* is a sub-procedure of *X*, then the *occurrence of D in C* is the same as the occurrence of *D* in *X*.

v. If *C* is identical to ${}^2X$ and *D* is a sub-procedure of *X*, then the *occurrence of D in C* is the same as the occurrence of *D* in *X*.

vi. The mode of occurrence of procedure *D* in *C* is only due to i)-v).

- Note that item (v) does not 'propagate' the effect of Double Execution inside *D*. If it did, some variables would have cropped up as free, as per the critical item (v) of the original Def. 2.15.

Towards TIL 2025

Definition 4 (*Free/bound variables*) Let *C* be a procedure with at least one occurrence of variable *x.*

- If the occurrence of *x* in *C* is in ***displayed*** mode, then this occurrence of *x* is o-***bound*** *in C*.
- Let the occurrence of *x* in *C* be in the executed mode and let $[\lambda x_1 \ldots x_m\ X]$ be a sub-procedure of *C*. Then, if this occurrence of *x* is a sub-procedure of *X* and *x* is identical with x_i, $1 \leq i \leq m$, then this occurrence of *x* is ***λ-bound*** *in C*.
- If the occurrence of *x* in *C* is neither 0-bound nor λ-bound in *C*, then this occurrence of *x is **free** in C*.

Towards TIL 2025

Definition 5 (*Correct substitution*) Let x be a variable and C, D procedures. If x is not free in C, then the *result of substituting D for x in C* is C. Hence, let x be free in C.

- If C is identical to x, then the result of the *substitution of D for x in C* is D.
- If C is identical with $[X\ X_1 \ldots X_m]$, then the result of *substitution of D for x is in C* is $[Y\ Y_1 \ldots Y_m]$, where $Y, Y_1, \ldots, Y_m$ are the results of substitution of D for x in $X, X_1, \ldots, X_m$, respectively.
- Let C be identical with $[\lambda x_1 \ldots x_m Y]$, $1 \leq i \leq m$. Then let y_i be x_i if x_i does not occur free in D; otherwise, let y_i be the first variable ranging over the same type as x_i such that y_i does not occur in Y and does not occur free in D, and it is distinct from $y_1, \ldots, y_{i-1}$. Then, the result of the *substitution of D for x in C* is $[\lambda y_1 \ldots y_m Z]$, where Z is the result of the substitution of D for x in the result of the substitution of y_i for x_i $(1 \leq i \leq m)$ in Y.
- If C is identical to ${}^2X$ where X is a procedure, then the *result of the substitution of D for x in C* is ${}^2Y$, where Y is the result of the substitution of D for x in X.

Compensation Principle

- Till now, we dealt with ***syntactic consistency***.
- However, we should prove that syntactic and ***semantic computation*** produce the same result, which is declared by the

Compensation Principle. Let *C* be a procedure. Then, for any valuation *v* and a procedure *D*, if *D* *v*-produces an entity *d*, then *C*(*D*/*x*) *v*-produces an entity *c* iff *C* *v*(*d*/*x*)-produces *c*.

$$\text{If } D \Rightarrow_v d, \text{ then } C(D/x) \Rightarrow_v c \text{ iff } C \Rightarrow_{v(d/x)} c$$

- *Since the valuation of variables that appear as free in the interim step of Double Execution can affect the result of semantic computation*, we must specify another restriction.
- If procedure *C* is ^2X, then ***X*** must ***v*-produce a *closed* procedure**.
- ***Proof by induction*** according to the rank of a procedure.

Kosterec (2024) proposes another solution. Roughly, he says:

Variables that crop up as free in the interim step of semantic computation of Double Execution ^2C ***are free*** in ^2C and in *C*, though they ***do not occur*** in *C*.

- Assumptions: $\boldsymbol{y \Rightarrow_v x}$; $\boldsymbol{x \Rightarrow_v k}$; $\boldsymbol{D \Rightarrow_v j}$;
- Kosterec: Since *x* is *v*-produced by *y*, it is (in this valuation *v*) free in *y* and in ^2y. Hence,

$$^2y(D/x) \Rightarrow_v \boldsymbol{j}$$

$$y \Rightarrow_{v(j/x)} x,\ x \Rightarrow_{v(j/x)} j.\ ^2y \rightarrow_{v(j/x)} \boldsymbol{j}$$

- This is a peculiar solution; well, it is a proposal of a valid ***evaluation strategy***.
- However, this strategy ***fails*** in case of ***empirical*** evaluation (recall 'If-then-else'), and it ***does not solve the problem of β-conversion***

Concluding remarks

- I have introduced an improved version of TIL, called TIL 2025, in which we solved various problems stemming from computing or executing the twin procedures of Trivialization and Double Execution.
- In TIL 2010, these problems arose from too liberal a definition of occurrence of variable free for substitution, which led to a flawed definition of correct substitution. In an effort to generalize the pertinent definitions in Tichý (1988), Duží et al. (2010) went too far.
- The authors defined as free those variables that crop up during the computation of a procedure but do not occur *syntactically*, or *structurally*, in the original procedure. This was the gist of the problem.
- β-conversion by value degenerated into a non-equivalent transformation, and the attempted proof of consistency, i.e. of the validity of the Compensation Principle, turned out to be flawed as well.

Concluding remarks

- ***These problems flew under the radar, because their pathological features did not show up in the applications of TIL***. However, no system facing inconsistencies in its foundations can issue any guarantee that its applications are trustworthy.
- I introduced the guarantee required by applying the restrictions needed to solve the problems.
- Moreover, I showed that the problem is not limited to TIL only. The common feature that systems operating with structured procedures share is a ***pair of interacting procedures or constructors that behave like Trivialization and Double Execution***. It is these twin procedures that must be properly correlated in order for the whole system to be consistent and the computations of procedures to be correct.

References

- Duží, M., Jespersen, B. (2025): Twin procedures, two kinds of variable binding, and two kinds of computation. *Annals of Pure and Applied Logic*.
- Pezlar, I. (2025). Hyperintensions as Computations. *Journal of Philosophical Logic* 54, 995–1018.
- Moggi, E. (1991). Notions of computation and monads. *Information and Computation* 93, 55–92.
- Kosterec, M. (2021). Substitution inconsistencies in Transparent Intensional Logic. *Journal of Applied Non-Classical Logics* 31, 355-371.
- Duží, M. (2025). Dual TIL as a logical experiment in a procedural λ-calculus. *Synthese*, forthcoming.

Thank you for your attention