The Algorithm of Context Recognition in TIL

Marie Duží, Michal Fait

VSB-Technical University Ostrava, Department of Computer Science FEI,
17. listopadu 15, 708 33 Ostrava, Czech Republic
marie.duzi@vsb.cz, michal.fait@vsb.cz

Abstract. The goal of this paper is to introduce the algorithm of context recognition in the functional programming language *TIL-Script*. The TIL-Script language is an operationally isomorphic syntactic variant of Tichý's Transparent Intensional Logic (TIL). From the formal point of view, TIL is a hyperintensional, partial, λ-calculus with *procedural* semantics. Due to ramified hierarchy of types it is possible to distinguish three levels of abstraction at which TIL constructions operate. At the highest *hyperintensional* level the object to operate on is a *construction* (though a higher-order construction is needed to present this lower-order construction as an object of predication). At the middle *intensional* level the object to operate on is the *function* presented, or constructed, by a construction, while at the lowest *extensional* level the object to operate on is the *value* (if any) of the presented function. Thus, a necessary condition for the development of an inference machine for the *TIL-Script* language is recognizing a context in which a construction occurs, namely extensional, intensional and hyperintensional context, so that inference rules can be properly applied.

Key words: Transparent Intensional Logic, TIL-Script, three kinds of context, context-recognition algorithm

1 Introduction

The family of automatic theorem provers, known today as HOL, is getting increasingly interest in logic, mathematics, and computer science.[1] These tools are broadly used in automatic theorem checking and applied as interactive proof assistants. As 'HOL' is an acronym for higher-order logic, the underlying logic is usually a version of a simply typed λ-calculus. This makes it possible to operate both in extensional and intensional contexts, where a value of the denoted function or the function itself, respectively, is an object of predication.

Yet there is another application that is gaining interest, namely natural-language processing. There are large amount of text data that we need to analyse and formalize. Not only that, we also want to have question-answer systems which would infer implicit computable knowledge from these large explicit knowledge bases. To this end not only intensional but rather hyperintensional logic is needed, because we need to formally analyse natural language

[1] See, for instance, [1] or [6].

in a fine-grained way so that the underlying inference machine is neither over-inferring (that yields inconsistencies) nor under-inferring (that causes lack of knowledge). We need to properly analyse agents' attitudes like knowing, believing, seeking, solving, designing, etc., because attitudinal sentences are part and parcel of our everyday vernacular. And attitudinal sentences, *inter alia*, call for a hyperintensional analysis, because substitution of a logically equivalent clause for what is believed, known, etc. may fail. Hyperintensional individuation is frequently also referred to as 'fine-grained' or sometimes simply 'intensional' individuation, when 'intensional' is not understood in the specific sense of possible-world semantics or in the pejorative sense of flouting various logical rules of extensional logic.

A principle of individuation qualifies as hyperintensional as soon as it is finer than necessary equivalence. The main reason for introducing hyperintensionality was originally to block various inferences that were argued to be invalid. The theoretician introduces a notion of hyperintensional context, in which the proper substituends are hyperintensions rather than the modal intensions of possible-world semantics or extensions. For instance, if Tilman is solving the equation $sin(x) = 0$, then he is not solving the infinite set of multiples of the number π, because this set is the solution and there would be nothing to solve, the solver would immediately be a finder. Yet there is something Tilman is solving. He is trying to execute the procedure specified by $\lambda x.sin(x)=0$. Thus, there is the other side of the coin, which is the positive topic of which inferences should be validated in hyperintensional contexts.

TIL definition of hyperintensionality is positive rather than negative. Any context in which the meaning of an expression is *displayed* rather than *executed* is hyperintensional.[2] Moreover, our conception of meaning is *procedural*. Hyperintensions are abstract procedures rigorously defined as TIL constructions which are assigned to expressions as their context-invariant meanings. This entirely anti-contextual and compositional semantics is, to the best of our knowledge, the only one that deals with all kinds of context, whether extensional, intensional or hyperintensional, in a uniform way. The same extensional logical laws are valid invariably in all kinds of context. In particular, there is no reason why Leibniz's law of substitution of identicals, and the rule of existential generalisation were not valid. What differ per the context are not the rules themselves but the types of objects on which these rules are applicable. In an *extensional* context they are *values* of the functions denoted by the respective expressions; in an *intensional context* the rules are applicable on the denoted *functions* themselves, and finally in a *hyperintensional context* the *procedures* that is the meanings themselves are the objects to operate on. Due to its stratified ontology of entities organised in a ramified hierarchy of types, TIL is a logical

[2] In [2] we use the terms 'mentioned' vs. 'used'. But since these terms are usually understood linguistically as using and mentioning *expressions*, whereas in TIL we use or mention their *meanings*, here we vote for 'displayed' vs. 'executed', respectively. See also [3].

framework within which such an extensional logic of hyperintensions has been introduced.[3]

The *TIL-Script* language is an operationally isomorphic syntactic variant of TIL. The development of its inference machine is based on these principles. First, we implement the algorithm of *context recognition* that makes it possible to determine the type of an object to operate on. Second, we implement TIL *substitution method* (including β-conversion 'by value') that makes it possible to operate on displayed constructions in a hyperintensional context. Finally, we are going to implement a *hyperintensional variant of the sequent calculus* for TIL that has been specified in [4] and [8].

The goal of this paper is the description of the first step, that is of the context-recognition algorithm. The rest of the paper is organised as follows. In Section 2 we introduce basic principles of the TIL-Script language. The algorithm of context recognition is described in Section 3 and concluding remarks can be found in Section 4.

2 Basic Principles of TIL and the TIL-Script language

The TIL syntax will be familiar to those who are familiar with the syntax of λ-calculi with four important exceptions. *First*, TIL λ-terms denote abstract procedures rigorously defined as *constructions*, rather than the set-theoretic functions produced by these procedures.[4] Thus the construction *Composition* symbolised by $[FA_1 \dots A_m]$ is the very procedure of applying a function presented by F to an argument presented by $A_1, \dots, A_m$, and the construction *Closure* $[\lambda x_1 x_2 \dots x_n C]$ is the very procedure of constructing a function by λ-abstraction in the ordinary manner of λ-calculi. *Second*, objects to be operated on by complex constructions must be supplied by atomic constructions. Atomic constructions are one-step procedures that do not contain any other constituents but themselves. They are *variables* and *Trivialization*. Variables construct entities of the respective types dependently on valuation, they v-construct. For each type there are countably many variables assigned that range over this type (v-construct entities of this type). Trivialisation $'X$ of an entity X (of any type even a construction) constructs simply X. In order to operate on X, the object X must be grabbed first. Trivialisation is such a one-step grabbing mechanism. *Third,* since the product of a construction can be another construction, constructions can be executed twice over. To this end we have *Double Execution* of X, ^2X, that v-constructs what is v-constructed by the product of X. Finally, since we work with partial functions, constructions can be v-improper in the sense of failing to v-construct an object for a valuation v.[5]

[3] See, for instance,[4].

[4] For details see [9] and [3].

[5] TIL is one of the few logics that deal with partial functions, see also [7]. There are two basic sources of improperness. Either a construction is not type-theoretically coherent, or it is a procedure of applying a function f to an argument a such that f is not defined

Since TIL has become a well-known system, see, for instance [2], [9], and other numerous papers, in what follows we introduce only the grammar of the TIL-Script language, and characterize the syntactic differences between TIL and TIL-Script. The TIL-Script functional declarative language is a computational variant of TIL. It covers all the functionalities specified in the TIL system but slightly differs in its notation that applies purely the ASCII code. Thus, the syntax does not involve subscripts and superscripts, Greek characters, and special characters like '$\forall$'or '$\exists$'. These special symbols have been replaced by the key-words like 'Exist', 'ForAll', 'Bool', 'Time', 'World'. Greek 'λ' in Closure is replaced by '\'. The abbreviated '$\alpha_{\tau\omega}$' is in TIL-Script written as 'α@tw'. On the other hand, the set of basic data types is richer here to cover those useful in functional programming. In TIL-Script we distinguish the types of real and natural numbers from discrete times, and we also have the type String. Higher-order types $*_n$ are just $*$, we do not mark up the order of constructions, because it is controlled by the syntactic analyzer. In TIL-Script we also work with the functional type of a *List*. This type is defined by the key-word List and the types of its elements. For instance, List (Real) is a list of real numbers. Though this is a derived type, because each list can be defined as a function mapping natural numbers to the respective types, in practice it is much more convenient to work directly with the type List. The differences in basic types are summarized in Table 1.

Table 1: *TIL-Script* basic types

TIL	TIL-Script	*Description*
o	Bool	*Truth-values*
ι	Indiv	*Individuals (universe)*
τ	Time	*Times*
ω	World	*Possible worlds*
-	Int	*Integers*
τ	Real	*Real numbers*
-	String	*String of characters*
α	Any	*Unspecified type*
$*_n$	$*$	*Constructions of order n*

Here is the TIL-Script grammar. It is easy to check that this grammar specifies the language functionally isomorphic to the language specified in the classical TIL.

```
start = {sentence};
sentence = sentence content , termination;
```

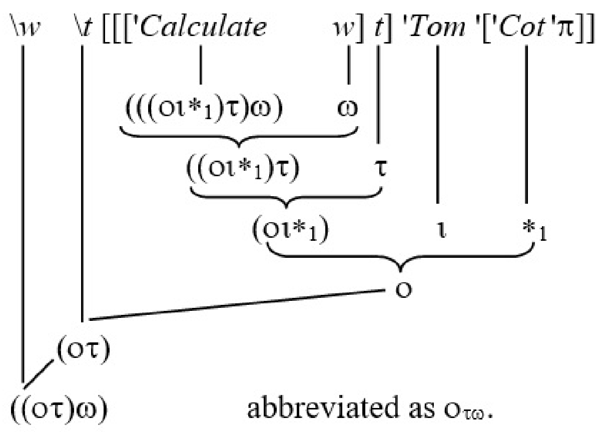
at a. For instance, Composition $[{}'Cotg\ {}'\pi]$ is v-improper for any valuation v, because the function cotangent is not defined at the number π in the domain of real numbers. *Single Execution* ${}^1X$ is improper for any valuation v in case X is not a construction.

```
sentence content = type definition | entity definition |
    construction | global variable definition;
termination = optional whitespace,".", optional whitespace;

type definition = "TypeDef", whitespace, type name, optional
    whitespace, ":=", optional whitespace, data type;
entity definition = entity name, {optional whitespace,",",
    optional whitespace, entity name}, optional whitespace,
    "/", optional whitespace, data type;
global variable definition = variable name, {optional
    whitespace, ",", optional whitespace, variable name},
    optional whitespace, "->", optional whitespace, datatype;
construction = (trivialisation | variable | composition |
    closure | n-execution) [, "@wt"];

data type = (embeded type | list type | touple type | user
    type | enclosed data type) [, '@tw'];
embeded type = "Bool" | "Indiv" | "Time" | "String" | "World"
     | "Real" | "Int" | "Any" | "*";
list type = "List", optional whitespace, "(", optional
    whitespace, data type, optional whitespace, ")";
touple type = "Tuple", optional whitespace, "(", optional
    whitespace, data type, optional whitespace, ")";
user type = type name;
enclosed data type = "(", optional whitespace, data type, {
    whitespace, data type}, optional whitespace ")";

variable = variable name;
trivialisation = "'", optional whitespace, (construction |
    entity);
composition = "[", optional whitespace, construction,
    optional whitespace, construction, {construction},
    optional whitespace, "]";
closure = "[", optional whitespace, lambda variables,
    optional whitespace, construction, optional whitespace,
    "]";
lambda variables = "\", optional whitespace, typed variables;
typed variables = typed variable, {optional whitespace,",",
    typed variable};
typed variable = variable name, [optional whitespace, ":",
    optional whitespace, data type];
n-execution = "^", optional whitespace, nonzero digit,
    optional whitespace, (construction | entity);

entity = keyword | entity name | number | symbol;

type name = upperletter name;
entity name = upperletter name;
variable name = lowerletter name;
```

```
keyword = "ForAll" | "Exist" | "Every" | "Some" | "True" | "
    False" | "And" | "Or" | "Not" | "Implies";
lowercase letter = "a" | "b" | ... | "z";
uppercase letter = "A" | "B" | ... | "Z";
symbols = "+" | "-" | "*" | "/";
zero = "0";
nonzero digit = "1" | "2" | ... | "9";
number = ( zero | nonzero digit),  { zero | nonzero digit }
    [".",  (zero | nonzero digit), { zero | nonzero digit }];
upperletter name = uppercase letter, { lowercase letter |
    uppercase letter | "_" | zero | nonzero digit };
lowerletter name = lowercase letter, { lowercase letter |
    uppercase letter | "_" | zero | nonzero digit };
whitespace = whitespace character, optional whitespace;
optional whitespace = { whitespace character };
whitespace character = ? space ? | ? tab ? | ? newline ?;
```

To illustrate TIL-Script analysis, we adduce an example of the analysis of the sentence "Tom calculates cotangent of the number π" followed by its derivation tree with type assignment. In the interest of better readability and a clear arrangement of the derivation tree, we use here Greek letters for types, as it is in classical TIL. According to the above grammar, the types are as follows: ο= Bool, ι= Indiv, τ= Time, ω= World, $o_{\tau\omega}$ = Bool@tw, $*_n = *$.

$$\backslash w \backslash t[[['Calculate\ w]\ t]\ 'Tom\ '['Cot'\pi]]$$



The resulting type is $o_{\tau\omega}$, that is the type of the proposition that Tom calculates Cotangent of π. The types of the objects constructed by 'π, '*Cot* and ['*Cot* 'π], that is τ, (ττ) and τ, respectively, are irrelevant here, because these constructions are not constituents of the whole construction. They occur only displayed by Trivialization '['*Cot* 'π], that is hyperintensionally. We are going to deal with this issue in the next section.

3 Context Recognition

The algorithm of context recognition is based on definitional rules presented in [2], §2.6. Since these definitions are rather complicated, here we introduce just the main principles. TIL operates with a fundamental dichotomy between hyperintensions (procedures) and their products, i.e. functions. This dichotomy corresponds to two fundamental ways in which a construction (meaning) can occur, to wit, *displayed* or *executed*. If the construction is displayed, then the procedure itself becomes an object of predication; we say that it occurs *hyperintensionally*. If the construction occurs in the execution mode, then it is a constituent of another procedure, and an additional distinction can be found at this level. The constituent presenting a function may occur either *intensionally* or *extensionally*. If intensionally, then the whole function is an object of predication; if extensionally, then a functional value is an object of predication. The two distinctions, between displayed/executed and intensional/extensional occurrence, enable us to distinguish between the three kinds of context:

– *hyperintensional context*: a construction occurs in a displayed mode (though another construction at least one order higher needs to be executed to produce the displayed construction)
– *intensional context*: a construction occurs in the executed mode to produce a function but not its value (moreover, the executed construction does not occur within another hyperintensional context)
– *extensional context*: a construction occurs in the executed mode in order to produce particular value of a function at a given argument (moreover, the executed construction does not occur within another intensional or hyperintensional context).

The basic idea underlying the above trifurcation is that the same set of logical rules applies to all three kinds of context, but these rules operate on different complements: procedures, produced functions, and functional values, respectively. A substitution is, of course, invalid if something coarser-grained is substituted for something finer-grained.

The algorithm of context recognition is realized in the Prolog programming language.[6] In the phase of the syntactic analysis of the TIL-Script language, a Prolog database of constructions and types is created. This database consists of three main parts:

1. *Typed objects* are pairs (name, type), represented as binary relations (type/2).
2. *Typed* global *variables* are pairs (name, type) represented as binary relations (globalVariable/2).
3. *Constructions*; constructions are the most complicated case. They are represented as 10-ary relations (construction/10);

[6] Context recognition system download is available here: `http://elogika.vsb.cz/projects/GA15-13277S/`.

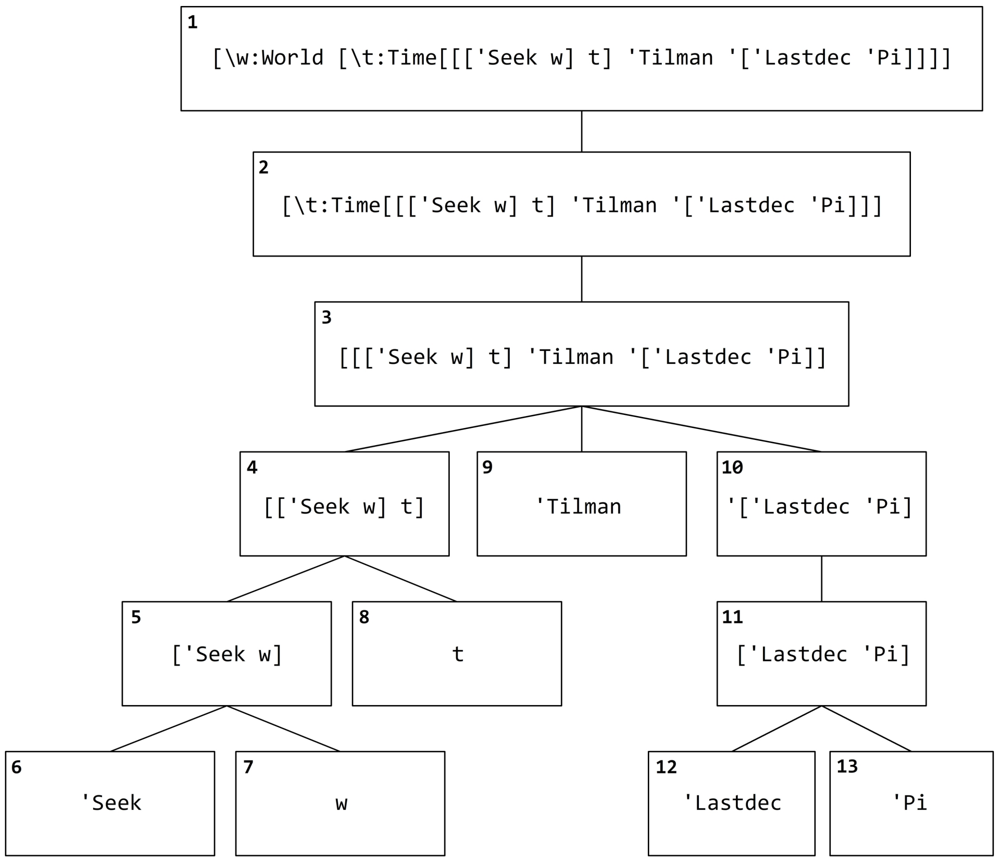

1
[\w:World [\t:Time[[['Seek w] t] 'Tilman '['Lastdec 'Pi]]]]
2
[\t:Time[[['Seek w] t] 'Tilman '['Lastdec 'Pi]]]
3
[[['Seek w] t] 'Tilman '['Lastdec 'Pi]]
4
[['Seek w] t]
9
'Tilman
10
'['Lastdec 'Pi]
5
['Seek w]
8
t
11
['Lastdec 'Pi]
6
'Seek
7
w
12
'Lastdec
13
'Pi

Fig. 1: Derivation tree of the construction

Each construction or subconstruction denoted by a term of an input TIL-Script file is recorded as a relation construction/10. For example, having the term '[\x ['+ '5 '4]]', five records of construction/10 are created, because five constructions are denoted here; they are [\x ['+ '5 '4]], ['+ '5 '4], '+, '5 and '4. Each record contains a unique identifier ID, which can be referred to by another construction. In this way we obtain a derivation tree structure of each construction. Figure 1 illustrates the tree structure of the construction assigned to the sentence "Tillman is seeking last decimal of the number π." As it is common in Prolog, the algorithm applies the depth-first search strategy with backtracking. The numbers of nodes in Figure 1 illustrate this strategy. The algorithm recursively calls itself for every child node or terminates if the current node is a leaf (that is a construction record without a child ID list). Whenever a leaf node is reached, backtracking comes into the scene and another branch is being searched.

The algorithm of context recognition consists of several steps. First, we recognize hyperintensional occurrences of displayed constructions, that is those that occur in the scope of a Trivialization, the effect of which, however, has not been cancelled by Double Execution, because $^{2}{}'C$ is equivalent to C. Moreover, a higher-level context is dominant over a lower-level one. Thus, if C occurs in D hyperintensionally, then all the subconstructions of C occur hyperintensionally in D as well. Here is the algorithm to determine hyperintensional occurrences.

```
Name: determineHyperintensional
Input: construction C, constant S indicating whether a current
construction occurrence is displyed or executed; in the beginning
S is set to unknown.

If S=’’mentioned’’
    Save hyperintensional context of construction C
If S=’’used’’ and C is trivialisation of another construction ‘D
    call determineHyperintensional(D,"mentioned")
else
    If S=’’used’’ and C is double execution ^2D and D is
    trivialization of a construction 'E
        call determineHyperintensional(E,"used")
    Else
        P = child nodes of construction C
        For every construction X in P do
            determineHyperintensional (X,S)
```

If the occurrence of C within D is not hyperintensional, then it occurs in the execution mode as a constituent of D, and the object that C v-constructs (if any) plays the role of an argument to operate on. In such a case we have to distinguish whether C occurs *intensionally* or *extensionally*. To this end we first distinguish between extensional and intensional supposition of C. Since C occurs executed, it is typed to v-constructs a function f of type $(\alpha\beta_1 \dots \beta_n)$, n possibly equal to zero. Now C may be composed within D with constructions $D_1 \dots D_n$ which are typed to v-construct the arguments of the function f, that is Composition $[CD_1 \dots D_n]$ is a constituent of D. In such a case we say that C occurs in D with *extensional supposition*. Otherwise C occurs in D with *intensional supposition* that is intensionally.

The algorithm first determines occurrences with extensional supposition, and then it is in a position to check whether a given construction that occurs with extensional supposition is, or is not occurring within a λ-generic context. If the context is non-generic, then the respective occurrence is extensional. Otherwise, the context is intensional. Here is the algorithm for extensional-supposition recognition.

```
Name: extSupositionConstructions
Output: set of constructions with extensional suposition

For every construction C do
If C is composition [X Y1...Ym]
    If C does not occur in hyperintensional context
```

```
        Add construction X to the result
If C is execution ^1X or ^2X, where X is object of order one
    If C does not occur in hyperintensional context
        Add construction C to the result
For every execution C in form ^2X, where X v-constructs object of
order one
    If C does not occur in hyperintensional context
        Add construction C to the result
```

The algorithm checking λ-genericity is specified as per Def. 11.6 of [5]. In principle, it checks whether to each Closure there is a pairing Composition. This is so because the procedure of constructing a function by λ-abstraction raises the context up to the intensional level, while the dual procedure of applying a function to an argument decreases the context down. The algorithm examines the derivation tree of a construction and dynamically creates a generic-type list that determines the genericity level. If the list is empty, the context is non-generic, otherwise it is λ-generic.

For example, the generic-type list of the Trivialisation '+ occurring in the following constructions is as follows:

– in Composition ['+ x y] the list is empty, hence non-generic context;
– in Closure [\x:Real ['+ x y]] the context is [[Real]]-generic;
– in Closure [\y:Real [\x:Real ['+ x y]]] the context is [[Real],[Real]]-generic;
– in Composition [[\y:Real [\x:Real ['+ x y]]] '5] the context is again [[Real]]-generic;
– in Composition [[[\y:Real [\x:Real ['+ x y]]] '5] '5] the list is empty, hence non-generic context;
– in Closure [\x:Real, y:Real ['+ x y]] the context is [[Real,Real]]-generic.

The algorithm for generic type recognition is specified as follows:

```
Name: genericity
Input: constructions D and C, where D is constituent C
Output: generic type

1.If C is atomic construction and C = C
      result = non-generic type
2.if C is closure [\x1,...,xm X]; x1 $\to_v$ γ1, ..., xm $\to_v$ γm, then:
      a)If D = C,
            β = genericity(X,X)
            result = ((γ1,...,ym)β)
      b)If D is constituent X,
            β = genericity(D,X)
            result = ((y1,...,ym)β)
3.If C is composition [X Y1...Ym]; Y1 $\to_v$ γ1,...,Ym $\to_v$ γm.
      a)If D=C
            result = genericity(X,C)
      b)If D is constituent X
             G=genericity(X,X)
             If G is non-generic type
                 result=genericity(D,X)
             Else if G is generic type  ((γ1,...,γm)β)
                 result = β
      c)If D is constituent of one construction Y from Yi
                result = genericity(D,Y)
4.If C is execution ^2X or ^1X where X is construction
      If D is constituent X
          result = genericity(D,X)
```

The last step of the context-recognition algorithm is easy. For every construction, if the occurrence is not hyperintensional or extensional, the result is intensional context.

```
Name: determineIntensional
    For every construction C
        If C does not occur intensionally neither hyperintensionally
            Save intensional context of construction C
```

Here is an example of the result of the syntactic analysis including context-recognition of the construction

[\w: World [\t:Time ['Seek@wt 'Tilman '['Lastdec 'Pi]]]].

```
<construction occurrence ="Intensional"
              construction ="[\w:World [\t:Time [[['Seek w] t] 'Tilman '['Lastdec 'Pi]]]]">
  <construction occurrence ="Intensional" construction ="[\t:Time [[['Seek w] t] 'Tilman '['Lastdec 'Pi]]]">
    <construction occurrence ="Intensional" construction ="[[['Seek w] t] 'Tilman '['Lastdec 'Pi]]">
      <construction occurrence ="Intensional" construction ="[['Seek w] t]">
        <construction occurrence ="Intensional" construction ="['Seek w]">
          <construction occurrence ="Intensional" construction ="'Seek"></construction>
          <construction occurrence ="Intensional" construction ="w"></construction>
        </construction>
        <construction occurrence ="Intensional" construction ="t"></construction>
      </construction>
      <construction occurrence ="Intensional" construction ="'Tilman"></construction>
      <construction occurrence ="Intensional" construction ="'['Lastdec 'Pi]">
        <construction occurrence ="Hyperintensional" construction ="['Lastdec 'Pi]">
          <construction occurrence ="Hyperintensional" construction ="'Lastdec"></construction>
          <construction occurrence ="Hyperintensional" construction ="'Pi"></construction>
        </construction>
      </construction>
    </construction>
  </construction>
</construction>
```

4 Conclusion

We introduced the computational variant of Transparent Intensional Logic, the TIL-Script functional programming language. Our main novel result is the implementation of the algorithm that recognises three kinds of context, namely extensional, intensional and hyperintensional, which is a necessary condition for the implementation of the TIL-Script inference machine. It is an important result, because when testing the algorithm, it turned out that there are still slight unintended inaccuracies in the definitions as presented in [5], which in turn led to their revision.

Acknowledgments. This work has been supported by the Grant Agency of the Czech Republic project No. GA15-13277S, "Hyperintensional logic for natural language analysis" and by the internal grant agency of VSB-Technical University Ostrava, project No. SP2016/100, "Knowledge modelling and its applications in software engineering II".

References

1. Benzmüller, Ch. (2015): Higher-Order Automated Theorem Provers. In *All about Proofs, Proof for All*, David Delahaye, Bruno Woltzenlogel Paleo (eds.), College Publications, Mathematical Logic and Foundations, pp. 171-214.
2. Duží, M., Jespersen B., Materna P. (2010): *Procedural Semantics for Hyperintensional Logic; Foundations and Applications of Transparent Intensional Logic*. Berlin, Heidelberg: Springer.
3. Duží, M., Jespersen, B. (2015): Transparent Quantification into Hyperintensional objectual attitudes. Synthese, vol. 192, No. 3, pp. 635-677.
4. Duží, M. (2012): Extensional logic of hyperintensions. *Lecture Notes in Computer Science*, vol. 7260, pp. 268-290.
5. Duží, M., Materna P. (2012): *TIL jako procedurální logika.* Aleph Bratislava.
6. Gordon, M. J. C., Melhan T. F. (eds).: *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.
7. Moggi, E. (1988): *The Partial Lambda-Calculus*, PhD thesis, University of Edinburg, available as LFCS report at http://www.lfcs.inf.ed.ac.uk/reports/88/ECS-LFCS-88-63/.
8. Tichý, P. (1982): Foundations of partial type theory. *Reports on Mathematical Logic*, vol. 14, pp. 52-72. Reprinted in (Tichý 2004: 467-480).
9. Tichý, P. (1988):*The Foundations of Frege's Logic*. Berlin, New York: De Gruyter.
10. Tichý, P. (2004): *Collected Papers in Logic and Philosophy*, eds. V. Svoboda, B. Jespersen, C. Cheyne. Prague: Filosofia, Czech Academy of Sciences, and Dunedin: University of Otago Press.