A Lambda Term Representation Inspired by Linear Ordered Logic

Andreas Abel
Theoretical Computer Science
Institut für Informatik
Ludwig-Maximilians-Universität
München, Germany
andreas.abel@ifi.lmu.de

Nicolai Kraus
Functional Programming Laboratory
School of Computer Science
University of Nottingham
Nottingham, United Kingdom
ngk@cs.nott.ac.uk

We introduce a new nameless representation of lambda terms inspired by ordered logic. At a lambda abstraction, number and relative position of all occurrences of the bound variable are stored, and application carries the additional information where to cut the variable context into function and argument part. This way, complete information about free variable occurrence is available at each subterm without requiring a traversal, and environments can be kept exact such that they only assign values to variables that actually occur in the associated term. Our approach avoids space leaks in interpreters that build function closures.

In this article, we prove correctness of the new representation and present an experimental evaluation of its performance in a proof checker for the Edinburgh Logical Framework.

Keywords: representation of binders, explicit substitutions, ordered contexts, space leaks, Logical Framework.

1 Introduction

Type checking dependent types in languages like Agda [21] and Coq [14] or logical frameworks like Twelf [22] requires a large amount of evaluation, since types may depend on values. Such type checkers incorporate an interpreter for purely functional programs with free variables—at least, the λ-calculus—which is used to compute weak head normal forms of types. Efficiency of type checking is mostly identical with efficiency of evaluation[1] (and, in case of type reconstruction, efficiency of unification), and remains a challenge as of today. In seminal work, Gregoire and Leroy [13] have sped up Coq type checking by compiling to byte-code instead of using an interpreter. Boespflug [6] has obtained further speed-ups by producing native code using stock-compilers.

While compilation approaches are successful on batch type *checking* fully explicit programs, they have not been attempted on type *reconstruction* using higher-order unification or on interactive program construction such as in Agda and Epigram [18]. These languages are involved and constantly evolving, and their implementations are prototypes and frequently modified and extended. Implementing a full compiler just to get type reconstruction going is deterring; furthermore, compilation has not (yet) proven its feasibility in minor evaluation tasks (like weak head evaluation) that dominate higher-order unification. At least for language prototyping, smart interpreters are, and may remain, competitive with compilation.

For instance, Twelf's interpreter is sufficiently fast; it is inspired by a term representation with de Bruijn indices [7] and explicit substitutions [1]. In the context of functional programming, explicit substitutions are known as *closures*, consisting of the code of a function plus an environment, assigning

[1] Evaluation is necessary to reduce types to weak head normal form and compare types for equality. Subtracting these operations, type checking has linear complexity.

H. Geuvers, G. Nadathur (Eds): LFMTP 2011
EPTCS 71, 2011, pp. 1–13, doi:10.4204/EPTCS.71.1

values to the free variables appearing in the code. In typical implementations of interpreters [9], these environments are not precise; they assign values to all variables that are statically in scope rather than only to those that are actually referred to in the code. This bears potential for space-leaks: the environment of a closure might refer to a large value that is never used, but cannot be garbage collected. An obvious remedy to this threat is, when forming a closure, to restrict the environment to the actual free variables; however, this requires a traversal of the code. We explore a different direction: we are looking for a code representation that maintains information about the free variables at each node of the abstract syntax tree.

A principal candidate is *linear typing* in Curry-Howard correspondence with Girard's linear logic [12]; there, each variable in scope is actually referenced (more precisely, referenced exactly once). In other words, the free variables are exactly the variables in the typing context. Dropping types, we may talk of *linear scoping*. Yet we do not want to represent linear terms, but arbitrary λ-terms. Kesner and Lengrand [15] achieve this by introducing explicit term constructs for weakening and contraction. We pursue a different path: we incorporate information about variable use and multiplicity directly into abstraction and application.

In the context of linear λ-calculus, the free variables of a function application are the disjoint union of the free variables of the function and the free variables of the argument. If we want to maintain the set of free variables during a term traversal, at an application node we need to decide which variables go into the function part and which into the argument part. Thus, we would store at each application a set of variables that go into the, say, function part, all others would go to the argument part. Less information is needed if we switch to an *ordered* representation.

Ordered logic, also called *non-commutative linear logic* [23], refines linear logic by removing the structural rule *exchange* which restricts hypotheses to be used *in the order they have been declared*. Transferring this principle to ordered scoping this means that the scoping context lists the free variables in the order they occur in the term, from left to right. This allows pushing the context into an application with very little information: we just need to know *how many* variables appear in the function part so we can cut the context in two at the right position, splitting it into function context and argument context. This constitutes the central idea of our representation: at each application node of the syntax tree, we store a number denoting the number of free variable occurrences in the function part. During evaluation of an application in an environment, we can cut the environment into two, the environment needed for the evaluation of the function and the environment needed for the evaluation of the argument. Thus, our environments are precise and space leaks are avoided. In particular, a variable is always evaluated in a singleton environment assigning only a value to this variable. Following this observation, variables do not need a name, they are identified by their position; and environments are simply sequences of values.

Since we are not interested in proof terms of ordered logic per se, but only borrow the *ordered context* idea for our representation of untyped λ-calculus, we need to allow multiple occurrences of the same variable. In fact, the context shall list the variable *occurrences* in order. At a lambda abstraction, we bind all occurrences of the same variable. Thus, at an abstraction node we specify at which positions the bound variable should be inserted in the scoping context. This concludes the presentation of our idea.

In the rest of the paper, after an introductory example (Section 2) we formally define our term representation in Section 3. Interpreter and handling of environments are described in Section 4, followed by the translation between ordinary lambda terms and ordered terms (Section 5). Soundness of the interpreter is formally proven in Section 6, before we conclude with an experimental evaluation in Section 7.

This article summarizes the B.Sc. thesis of the second author [16].

2 An Example

To demonstrate the discussed risk of space-leaks during evaluation, we apply the term $\lambda x.\lambda y.a\,b\,y$ in basic syntax consecutively to the free variables g and f. A possible (and, if the mentioned closures are used, typical) sequence of reduction steps is given below. By writing $t\,[s_1/x_1,\ldots,s_n/x_n]$, we want to express that in the term t, each occurrence of the variable x_1 ($x_2,\ldots,x_n$) has to be replaced by the term s_1 (resp. $s_2,\ldots,s_n$) simultaneously. Such a substitution list always applies only to the directly preceding term:

$$\begin{aligned}
& (\lambda x.\lambda y.a\,b\,y)\,g\,f \\
\longrightarrow\quad & (\lambda y.a\,b\,y)[g/x]\;\; f \\
\longrightarrow\quad & (a\,b\,y)[g/x,f/y] \\
\longrightarrow\quad & (a\,b)\,[g/x,f/y]\;\; y\,[g/x,f/y] \\
\longrightarrow\quad & a\,[g/x,f/y]\;\; b\,[g/x,f/y]\;\; f \\
\longrightarrow\quad & a\,b\,f
\end{aligned}$$

Here, the substitution $[g/x]$ could be dropped instantly and there is no need to apply the other substitution $[f/y]$ to the term $a\,b$. However, the term representation used above comes along with the problem that such an evaluation algorithm does not have the required information in time. This is due to the fact that the binding information is always split between the λ and the actual variable occurrence, as they both carry the variable name. In contrast, using de Bruijn indices would make it possible to remove the piece of information from the λ. Our goal is to do it the other way round: We want the whole information to be available at the λ, thus making it possible to know the number and places of the bound variable occurrences without looking at the whole term.

3 Syntax

In this article, we only cover the core constructs of the lambda calculus as they are enough to make the approach clear. However, we do not see any limitations for common extensions. We first define ***ordered preterms***:

$$\begin{array}{llcll}
\text{ordered preterms} & \ni & t,u & ::= & x & \text{free variable (named } x\text{)} \\
 & & & \mid & \bullet & \text{bound variable (nameless)} \\
 & & & \mid & t\,{}^{m}\,u & \text{application} \\
 & & & \mid & \lambda^{\vec{k}}.t & \text{abstraction}
\end{array}$$

Free variables are denoted by their name like in the standard syntax. *Bound variables*, however, are just denoted by a dot $\bullet$, which does not carry any information beside the fact that it is a bound variable. In the case of an *application*, there is a first term (the function part) and a second term (the argument) as usual. Furthermore, the application carries an integer m as an additional piece of information that will be important for the evaluation process and is explained in a moment. The most interesting part is the *abstraction* $\lambda^{\vec{k}}.t$. The vector $\vec{k} = [k_1,k_2,\ldots,k_n]$ is nothing else but a list of non-negative integers of length n. It determines which dots $\bullet$ are bound by the λ in the following way: Consider all $\bullet$ in the term t which are not bound in t itself. Now, the first k_1 of these are not bound by the λ, the next one is, the following k_2 are again not bound and so on (see examples below).

We denote the number of unbound $\bullet$ in an ordered preterm t by $\mathsf{fV}(t)$. Consequently, $\mathsf{fV}(\cdot)$ is simply defined by:

$$\mathsf{fV}(x) = 0, \quad \mathsf{fV}(\bullet) = 1, \quad \mathsf{fV}(t\,{}^m u) = \mathsf{fV}(t) + \mathsf{fV}(u), \quad \mathsf{fV}(\lambda^{[k_1,\ldots,k_n]}.t) = \mathsf{fV}(t) - n.$$

We call an ordered preterm u an ***ordered term*** iff each sub-preterm w of u fulfils the following condition: If w is an application $t\,{}^m u$, the equation $m = \mathsf{fV}(t)$ holds and if it is an abstraction $\lambda^{[k_1,\ldots,k_n]}.t$, then $n + \sum_i k_i \leq \mathsf{fV}(t)$ is satisfied. The latter condition states that if a λ in u binds a variable, this variable must actually exist while the first one just gives a meaning to the integer carried by an application. Clearly, any sub-preterm of an ordered term is again an ordered term. u is called ***closed*** if $\mathsf{fV}(t) = 0$.

Here are some examples of closed terms. The S combinator

$$\lambda x.\,\lambda y.\,\lambda z.\,x z\,(y z)$$

would be written as

$$\lambda^{[0]}.\,\lambda^{[1]}.\,\lambda^{[1,1]}.\,\bullet^1\,\bullet^2\,(\bullet^1\,\bullet)$$

Moreover, the term

$$(\lambda x.\,\lambda y.\,a\,b\,y)\,g\,f$$

from Section 2 would be represented as

$$\left(\lambda^{[]}.\,\lambda^{[0]}.\,a^0\,b^0\,\bullet\right)\,{}^0\,f\,{}^0\,g$$

(note that applications are still left-associative). We can see that the first λ does not bind anything as it is annotated with the empty vector $[]$, while this is less obvious when it is written as λx.

At this point, we hope to have clarified the intended meaning of our syntax. A formal definition will be given in Section 5.

4 Values and Evaluation

Before specifying values and evaluation formally, we want to give an example to demonstrate how the information carried by a lambda should be used and why we always have exactly the needed information. Suppose we have the term

$$\left(\lambda^{[0]}.\,\lambda^{[1]}.\,\lambda^{[1,1]}.\,\bullet^1\,\bullet^2\,(\bullet^1\,\bullet)\right)\,g\,f\,n$$

that is, the S combinator applied to three free variables (we suppress the application indices 0 for better readability). We want to get rid of the beta redexes, so we start by eliminating the first one. The outermost λ is decorated with the vector $[0]$ of length one. Now, the single variable bound by this λ should be replaced by g, so we start a substitution list and insert a single g:

$$\left(\lambda^{[1]}.\,\lambda^{[1,1]}.\,\bullet^1\,\bullet^2\,(\bullet^1\,\bullet)\right)\,[g]\;\;f\,n$$

The first remaining λ is $\lambda^{[1]}$, so it does not bind the first variable (thus g remains first in the substitution list), but the second one. Consequently, we add an f after the g:

$$\left(\lambda^{[1,1]}.\,\bullet^1\,\bullet^2\,(\bullet^1\,\bullet)\right)\,[g,f]\;\;n$$

Now the situation becomes more interesting. The only remaining λ is now decorated with the vector $[1,1]$, so one n has to be inserted after the first entry (g) and another one must be placed after the subsequent entry (f):

$$\left(\bullet^1\,\bullet^2\,\left(\bullet^1\,\bullet\right)\right)[g,n,f,n]$$

All λ have now been eliminated. The applications' indices tell us how the substitution list should be divided between the terms:

$$\left(\bullet^1\,\bullet\right)[g,n]\;\;\left(\bullet^1\,\bullet\right)[f,n]$$

We do the same step once more:

$$\bullet[g]\;\;\bullet[n]\;\;(\bullet[f]\;\;\bullet[n])$$

The only thing left to be done is to apply the substitutions in the obvious way:

$$g\,n\,(f\,n)$$

Here, evaluation naturally leads to a term in beta normal form. This is not always the case: as an example, if we had tried to evaluate the above term without the n (i.e. $S^0\,f^0\,g$), we would have got stuck at $\left(\lambda^{[1,1]}.\bullet^1\,\bullet^2\,\left(\bullet^1\,\bullet\right)\right)[g,f]$. However, this would have been satisfactory as it would have shown that the term's normal form is an abstraction. In other words, our evaluation results in weak head normal forms.

Consequently, we define ***values*** in the following way:

$$\begin{array}{lcrll} \mathsf{values} & \ni & v,w & ::= & x\,\vec{v} & \text{large application} \\ & & & \mid & (\lambda^{\vec{k}}.t)^{\vec{v}} & \text{closure} \end{array}$$

The *large application*[2] consists of a variable x which is applied to a vector $[v_1,v_2,\ldots,v_m]$ of values. It is to be read as a left-associative application, i.e. as $((x\,v_1)\,v_2\ldots)\,v_m$. Note that it is not necessarily "large". Quite the contrary, it often only consists of the head (and the vector of values is empty).

A *closure* $(\lambda^{\vec{k}}.t)^{\vec{v}}$ is the result of the evaluation process if the corresponding beta normal form of the term does not start with a free variable. The main part, $\lambda^{\vec{k}}.t$, is nothing other than a lambda abstraction in the syntax of ordered terms. In addition, we need the substitution list $\vec{v}$ (which is simply a list of values) that satisfies $\mathrm{length}(\vec{v}) = \mathsf{fV}(\lambda^{\vec{k}}.t)$. The idea is that the i^{th} unbound $\bullet$ is to be replaced by v_i. These substitution lists have already been used in the example above.

At this point, we want to introduce a notation for inserting a single item multiple times into a list. More precisely, if $\vec{v} = [v_1,v_2,\ldots,v_m]$ is a list, $\vec{k} = [k_1,k_2,\ldots,k_n]$ is a vector (i.e. also a list) of non-negative integers satisfying $\sum_{i=1}^n k_i \leq m$ and w is a single item, we write $\vec{v}^{\,w/\vec{k}}$ for the list that is constructed by inserting w at each of the positions $k_1, k_1+k_2, \ldots, \sum_{i=1}^n k_i$ into $\vec{v}$, i.e. for the list $[v_1,v_2,\ldots,v_{k_1},w,v_{k_1+1},\ldots,v_{k_1+k_2},w,v_{k_1+k_2+1},\ldots,v_m]$ (of course, it is possible that $\vec{v}^{\,w/\vec{k}}$ starts or ends with w).

We are now able to define the evaluation function $[\![\cdot]\!]_{\cdot}$ which takes an ordered term t as well as an ordered substitution list $\vec{v}$ and returns a value. The tuple must always satisfy the condition $\mathsf{fV}(t) = \mathrm{length}(\vec{v})$. In other words, the list carries neither too little nor redundant information. At the start of the evaluation, the ordered substitution list is empty. Additionally, we specify the application $\cdot\,@\,\cdot$ of two values, which also returns a value and does not need anything else. Our evaluation procedure uses a call-by-value strategy:

[2] The argument vector $\vec{v}$ of a large application is sometimes called a *spine* [8]. Large applications $x\,\vec{v}$ also appear in the formulation of Böhm trees [4].

$$
\begin{aligned}
[\![x]\!]_{[]} &= x && (1)\\
[\![\bullet]\!]_{[v_1]} &= v_1 && (2)\\
[\![t^{\,k}\, u]\!]_{[v_1,\ldots,v_n]} &= [\![t]\!]_{[v_1,\ldots,v_k]} \mathbin{@} [\![u]\!]_{[v_{k+1},\ldots,v_n]} && (3)\\
[\![\lambda^{\vec{k}}.t]\!]_{\vec{v}} &= (\lambda^{\vec{k}}.t)^{\vec{v}} && (4)\\
(x\,\vec{v}) \mathbin{@} w &= x[\vec{v}, w] && (5)\\
(\lambda^{\vec{k}}.t)^{\vec{v}} \mathbin{@} w &= [\![t]\!]_{\vec{v}^{\,w/\vec{k}}} && (6)
\end{aligned}
$$

First, if we want to evaluate a free variable (1), the substitution list must be empty because of the invariant mentioned above. Second, in the case of a $\bullet$ (2), the ordered list must have exactly one entry. This entry is the result of the evaluation. If we evaluate an application (3), we evaluate the left and the right term. The application's index enables us to split the substitution list at the right position. Then, we have to apply the first result to the second. Evaluating an abstraction (4) is easy. We just need to keep the substitutions to build a closure.

If we want to apply a large application to a value w (5) , we just append w to the vector of values (we write $[\vec{v}, w]$ for $[v_1, \ldots, v_n, w]$). The case of a closure $(\lambda^{\vec{k}}.t)^{\vec{v}}$ (6) is less simple, but it is still quite clear what to do: $\vec{k}$ determines at which positions w should be inserted in the ordered substitution list, so we just construct the list $\vec{v}^{\,w/\vec{k}}$. Then, t is evaluated.

Concerning substitution lists, we talk about "lists of values" for simplicity. More specifically, we want them to be lists of pointers to avoid the duplication of "real" values during constructing lists like $\vec{v}^{\,w/\vec{k}}$. Instead of simple linked lists, we have also implemented these lists as dynamic functional arrays represented as binary trees. This reduces the asymptotic costs of list splitting—$[v_1, \ldots, v_n]$ to $([v_1, \ldots, v_k], [v_{k+1}, \ldots, v_n])$— and multi-insertion $\vec{v}^{\,w/\vec{k}}$ from linear to logarithmic time (in terms of the length of the list $\vec{v}$). For an experimental comparison of the two implementations see Section 7.

5 Parsing and Printing

In this section, we define how terms in normal syntax are translated into our ordered syntax (Parsing) and vice versa (Printing). To specify this, we need some notation. First of all, we write X for the set of variable names we want to use and T for the set of lambda terms in basic standard syntax (i.e. $X \subset T$, furthermore, $x \in X$ together with $t, u \in T$ implies $t\,u \in T$ and $\lambda x.t \in T$). Additionally, oT is the set of terms in our ordered syntax defined above, $\overline{oT}$ the subset of closed ordered terms ($\mathsf{fV}(t) = 0$) and V the set of values (defined in the previous section). Moreover, we write $\vec{X}$ for the set of lists of variable names and $\vec{V}$ for the set of lists of values. For each set Γ of variable names ($\Gamma \subseteq X$), we denote the set of lists of elements of Γ by $\vec{\Gamma}$ and the ordered terms that do not contain any variable of Γ (as a free variable) by oT_Γ.

By writing $oT \otimes \vec{X}$ (resp. $oT \otimes \vec{V}$, $oT_\Gamma \otimes \vec{\Gamma}$, ...), we mean the subset $\{(t, \vec{x}) \mid \mathsf{fV}(t) = \mathrm{length}(\vec{x})\}$ of $oT \times \vec{X}$ (and analogous for the other cases).

For a finite set Γ of variable names, we define the ***correspondence relation*** $\cdot \lhd \Gamma \rhd \cdot[\cdot] \subset T \times oT \times \vec{X}$ (pronounce: "corresponds in context Γ to"). The intuition is that $M \lhd \Gamma \rhd u[\vec{x}]$ means: M is a term that corresponds to the ordered term u, where unbound $\bullet$ are replaced by the (*not* necessarily pairwise distinct) variables in the list $\vec{x}$. The set Γ can be seen as a filter that tells us which free variables do not occur in u

but in $\vec{x}$ instead.

$$\frac{x \in \Gamma}{x \lhd \Gamma \rhd \bullet[x]} \quad (A) \qquad \frac{M \lhd \Gamma \rhd t[\vec{x}] \qquad N \lhd \Gamma \rhd u[\vec{y}] \qquad \mathrm{length}(\vec{x}) = m}{M\,N \lhd \Gamma \rhd (t^{\,m}\,u)[\vec{x},\vec{y}]} \quad (B)$$

$$\frac{x \notin \Gamma}{x \lhd \Gamma \rhd x[]} \quad (C) \qquad \frac{M \lhd (\Gamma \cup \{z\}) \rhd t[\vec{x}^{z/\vec{k}}] \qquad z \notin \vec{x}}{\lambda z.M \lhd \Gamma \rhd (\lambda^{\vec{k}}.t)[\vec{x}]} \quad (D)$$

It is important to note that $M \lhd \Gamma \rhd u[\vec{x}]$ implies that each variable occurring in $\vec{x}$ is contained in Γ and each free variable occurring in u is not contained in Γ. This can be shown by induction on M (simultaneously for all sets Γ).

By the same argument, one can see that (for each M and Γ) there exists a unique tuple $(u,\vec{x})$ satisfying $M \lhd \Gamma \rhd u[\vec{x}]$, so we can consider $(\lhd\Gamma\rhd)$ a function $T \to oT_\Gamma \times \vec{\Gamma}$. Furthermore, we note that $M \lhd \Gamma \rhd u[\vec{x}]$ always implies $\mathsf{fV}(u) = \mathrm{length}(\vec{x})$.

This also works the other way round. For each Γ and each tuple $(u,\vec{x}) \in oT_\Gamma \otimes \vec{\Gamma}$, there is (by induction on u) a term $M \in T$ satisfying $M \lhd \Gamma \rhd u[\vec{x}]$. Moreover, this term M is unique up to α equivalence. So, $(\lhd\Gamma\rhd)$ is actually a bijection between T/α (the set of α equivalence classes of terms) and $oT_\Gamma \otimes \vec{\Gamma}$. The inference rules above show how to apply this bijection or its inverse to a term or a tuple (in the last rule, any variable satisfying the condition can be chosen for z), so we have a computable bijection $T/\alpha \leftrightarrow oT_\Gamma \otimes \vec{\Gamma}$ determined by the rules for $(\lhd\Gamma\rhd)$.

Choosing $\Gamma = \emptyset$, we get a bijection $T/\alpha \leftrightarrow oT \otimes \vec{\emptyset}$. As $\vec{\emptyset}$ is only inhabited by the empty vector $[]$, we naturally get the ***parse*** function $\rhd$ which maps T/α bijectively on $\overline{oT}$.

The above construction also gives us a function $\overline{oT} \to T$, but this is not enough. We want to transform closures (elements of $oT \otimes \vec{V}$) and values (elements of V) into basic terms T. Therefore, we define the two ***print*** functions $\lll : oT \otimes \vec{V} \to T/\alpha$ and $\lhd : V \to T/\alpha$ simultaneously by recursion on the structure:

$$\begin{aligned}
\lll(x\ ,\ []) &= x && \text{(I)}\\
\lll(\bullet\ ,\ [v]) &= \lhd(v) && \text{(II)}\\
\lll(t^{\,m}u\ ,\ \vec{v}) &= \lll(t\ ,\ \vec{v}_{start}) \quad \lll(u\ ,\ \vec{v}_{rest}) && \text{(III)}\\
&\quad \text{(split } \vec{v} \text{ at position } m \text{ to get } \vec{v}_{start} \text{ and } \vec{v}_{rest}\text{)}\\
\lll\left(\lambda^{\vec{k}}.t\ ,\ \vec{v}\right) &= \lambda z.\lll\left(t\ ,\ \vec{v}^{z/\vec{k}}\right) && \text{(IV)}\\
&\quad \text{where } z \text{ is any variable that does not occur freely in } t \text{ or } \vec{v}\\
\lhd(x\,v_1\,v_2\,\ldots\,v_n) &= x \lhd(v_1) \lhd(v_2)\ \ldots\ \lhd(v_n) && \text{(V)}\\
&\quad \text{(a large application simply becomes an application of terms)}\\
\lhd\left((\lambda^{\vec{k}}.t)^{\vec{v}}\right) &= \lll\left(\lambda^{\vec{k}}.t\ ,\ \vec{v}\right) && \text{(VI)}
\end{aligned}$$

First, note that the printing functions are well-defined (i.e., they always terminate). This is because during evaluation of $\lll(u\ ,\ \vec{v})$, we may safely assume that $\lll(t\ ,\ \vec{w})$ is well-defined as long as t is a strict subterm of u and each value w' in $\vec{w}$ is either only a variable (so termination of $\lhd(w')$ is clear) or also in $\vec{v}$. Similar, during evaluation of $\lhd(x\,v_1\,\ldots\,v_n)$, we may assume that $\lhd(v_i)$ is defined for each i.

For all $t \in \overline{oT}, M \in T$, we have $\lll(t\ ,\ []) = M$ if and only if $M \lhd \emptyset \rhd t[]$ (which just means $\rhd t = M$) as both judgements are defined identically in the case of closed ordered terms. This essentially (with implicit use of the bijection $\overline{oT} \leftrightarrow oT \otimes \vec{\emptyset}$) means $\lll \circ \rhd = id_T$, i.e. the composition of parsing and printing is the identity.

6 Correctness and Termination properties

We still have not shown that our evaluation algorithm given in Section 4 does not change the meaning of terms. The combination of parsing, evaluating and printing should never result in a term that is not beta equivalent to the original term. We also want to show a limited termination property. To keep our argument simple, we just sketch the proofs and hope that the ideas become clear.

First, we attend to the correctness question. We need to convince ourselves that *rewriting* according to the rules of the functions $[\![\cdot]\!]_{\cdot}$ and @ does not cause an error. By *rewriting*, we mean one step of the *normal* or *leftmost outermost* evaluation. We have demonstrated this in the example at the beginning of Section 4. Printing should result in a term that is β equivalent to the term we get if we rewrite before printing. This basically means that, for each evaluation rule on the left hand side of the following table, we have to check that the equality on the right hand side holds:

$[\![x]\!]_{[]}$	$=$	x	$\lll(x\ ,\ [])$	$=_\beta$	$\lhd(x)$	(1)
$[\![\bullet]\!]_{[v]}$	$=$	v	$\lll(\bullet\ ,\ v)$	$=_\beta$	$\lhd(v)$	(2)
$[\![t^{\,k}u]\!]_{[\vec{v},\vec{w}]}$	$=$	$[\![t]\!]_{\vec{v}}\ @\ [\![u]\!]_{\vec{w}}$	$\lll\left(t^{\,k}u\ ,\ [\vec{v},\vec{w}]\right)$	$=_\beta$	$\lll(t\ ,\ \vec{v})\lll(u\ ,\ \vec{w})$	(3)
$[\![\lambda^{\vec{k}}.t]\!]_{\vec{v}}$	$=$	$(\lambda^{\vec{k}}.t)^{\vec{v}}$	$\lll\left(\lambda^{\vec{k}}.t\ ,\ \vec{v}\right)$	$=_\beta$	$\lhd\left((\lambda^{\vec{k}}.t)^{\vec{v}}\right)$	(4)
$(x\vec{v})\ @\ w$	$=$	$x[\vec{v},w]$	$\lhd(x\vec{v})\lhd(w)$	$=_\beta$	$\lhd(x[\vec{v},w])$	(5)
$(\lambda^{\vec{k}}.t)^{\vec{v}}\ @\ w$	$=$	$[\![t]\!]_{\vec{v}^{w/\vec{k}}}$	$\lhd\left((\lambda^{\vec{k}}.t)^{\vec{v}}\right)\lhd(w)$	$=_\beta$	$\lll\left(t\ ,\ \vec{v}^{w/\vec{k}}\right)$	(6)

Note that "rewriting using the evaluation rules" results in expressions which are, more or less, a mixture of elements of $oT\otimes\vec{V}$ and V. To be precise, such an expression is either in $oT\otimes\vec{V}$ or in V or a tuple (to read as simple application) of two expressions. The 1^{st}, 2^{nd} and 4^{th} rule turn an element of $oT\otimes V$ into a value, the 3^{rd} turns it in a tuple of two such elements, and the last two rules turn a tuple of two values into one value or element. Although we do not define it formally, it should be clear how those expressions can be printed by using the printing functions for ordered terms and values (if an expression is a tuple, just print the function part and the argument part separately). In fact, while the function $[\![\cdot]\!]_{\cdot}$ is formulated as a big step evaluation, the rewriting process can be understood as the corresponding small step (or one step) evaluation.

In the first five rows of the table, we only have to look at the definitions of the printing functions to see that the printed terms are not only β equivalent but also equal. The very last rule (6) requires closer examination: By rule (IV), the term $\lhd\left((\lambda^{\vec{k}}.t)^{\vec{v}}\right)$ is equal to $\lambda z.\lll\left(t\ ,\ \vec{v}^{z/\vec{k}}\right)$ for a (sufficiently) fresh variable z. Now, the definitions of the printing functions are "context free" in a way that guarantees that z occurs exactly length$(\vec{k})$ times (free) in $\lll\left(t\ ,\ \vec{v}^{z/\vec{k}}\right)$. Furthermore, replacing those occurrences by $\lhd(w)$ results in the term $\lll\left(t\ ,\ \vec{v}^{w/\vec{k}}\right)$. This means that, starting with $\lhd\left((\lambda^{\vec{k}}.t)^{\vec{v}}\right)\lhd(w)$, we have to use exactly one β reduction step to get the term $\lll\left(t\ ,\ \vec{v}^{w/\vec{k}}\right)$.

As we have already seen that the composition of parsing a term and printing it afterwards does not

change anything (up to α equivalence), we can now conclude that parsing, evaluating (a finite number of rewriting steps) and printing is equivalent to a number of β reduction steps.

Now we discuss termination. Obviously, our evaluation function $[\![\cdot]\!]_\cdot$ does not always terminate as some terms do not have a weak head normal form. However, $[\![\cdot]\!]_\cdot$ terminates whenever it is applied to $(\triangleright t)$ if the usual β reduction is strongly normalizing on t. The main consequence of this is that evaluation terminates for all well-typed terms. To prove this statement, assume that there is such a term $s_0 \in T$ so that the evaluation of $t_0 := \triangleright s_0$ does not terminate. Then, we get an infinite sequence $t_0, t_1, t_2, \ldots$ where t_{i+1} is the result of rewriting (a subexpression of) t_i using one of the evaluation rules. If we print $t_0, t_1, t_2, \ldots$, we get a sequence $s_0, s_1, s_2, \ldots$ of terms in T, where s_{i+1} is either (α) equal to s_i or arises from s_i in exactly one β reduction step. If β reduction is strongly normalizing on s_0, the sequence has to become constant at some point, i.e. $s_N = s_{N+1} = s_{N+2} = \ldots$ for some N. This implies that, after the first N rewriting steps, rule (6) is not used anymore. Define the *weight* $w(t)$ of an expression t to be 1, if the expression is just an element of $oT \otimes V$, to be 2, if it is a value of the *closure* type, to be $1 + 2^n + w(v_1) + w(v_2) + \ldots + w(v_n)$, if it is a value of the form $x\, v_1 v_2 \ldots v_n$ (i.e. a *large application*) and, if it is a tuple of two expressions, as the sum of both weights. Then, each of the rewritings that are induced by the first five lines in the table increase the weight of the expression, so we get $w(t_N) < w(t_{N+1}) < w(t_{N+2}) < \ldots$; however, as the total number of values (and tuples in $oT \otimes V$) is bound by the length of the term we get after printing t_N (or any t_{N+i}), the sequence is bounded, resulting in the required contradiction.

7 Experiments and Results

The specified term representation and evaluation have been implemented in Haskell. They have been used by a type checker to check large files of dependently typed terms of the Edinburgh Logical Framework which were kindly provided by Andrew W. Appel (Princeton University). To make this possible, an extended syntax has been used that includes Π-types, constants and definitions. It is straightforward to expand our evaluation algorithm to the extended syntax—for details consult the Bachelor's thesis of the second author [16]. The substitution lists have been implemented as simple Haskell lists, and also as balanced binary trees (following Adams [2]) for better asymptotic complexity. Both variants were evaluated for performance [referred to as *Ordered (trees)* and *Ordered (lists)*].

For comparison, the completely analogous algorithm for terms in extended basic syntax (i.e. T) has been used [*Simple Closures*]. Furthermore, we have tested a strategy that always evaluates completely (i.e. produces β normal forms) using Hereditary Substitution [*Beta Normal Values*] [25].

Our main test file `w32_sig_semant.elf` with a size of approximately 21 megabytes contains a proof described in [3]. We also tested smaller parts of this file, more precisely, the first 6000, 10,000 and 12,000 lines without the rest (named `6000.elf` and so on). Later terms tend to be larger, so the tests with fewer lines needed much less time.

All tests were executed on the same server `baerentatze.cip.ifi.lmu.de` working with a CPU of type *AMD Phenom II X4 B95* (only one core used, 3 GHz) and 8 GB system memory. The measurements of space and time consumption are given in the following tables (rounded average values). More specifically, *time* refers to the total time, including parsing the input file and transforming (if necessary) the representation to our ordered representation or to one that uses De Bruijn terms. However, in our tests, the transformation process only took a negligible amount of time. The time value does, however, not include any printing of the terms or the values (with printing, the total time increased significantly). *Space* refers to the peak space usage of the whole process.

6000.elf (file size: 3.8 MB)

	time (sec)	space (MB)
Ordered (trees)	18.9	1111
Ordered (lists)	18.6	1114
Simple Closures	18.5	1152
Beta Normal Values	27.6	2034

10000.elf (file size: 12.9 MB)

	time (sec)	space (MB)
Ordered (trees)	61.0	3230
Ordered (lists)	60.6	3237
Simple Closures	60.0	3302
Beta Normal Values	98.7	5878

12000.elf (file size: 17.8 MB)

	time (sec)	space (MB)
Ordered (trees)	84.3	5096
Ordered (lists)	83.8	5103
Simple Closures	83.6	5226
Beta Normal Values	137.7	8513

Unsurprisingly, beta normal values perform significantly worse than each of the other possibilities. However, the difference is smaller than it could have been expected. This might be due to the fact that during type checking, total evaluation of a term is often necessary anyway, thereby reducing the hereditary substitution's disadvantage.

Although none of the other strategies exhibited any shortcomings in the comparisons above, the following results for the complete file are remarkable. Here, implementing ordered substitutions as normal Haskell lists seems to be much more efficient than using tree structures:

w32_sig_semant.elf (file size: 20.9 MB)

	time (sec)	space (MB)
Ordered (trees)	108.4	8877
Ordered (lists)	94.8	4948
Simple Closures	94.3	5068
Beta Normal Values	169.8	9044

Our Simple Closures are still on the same level as Ordered Representation with lists, but the trees are far behind. In comparison, the type checker of the Twelf project, *Twelf r1697* (written in *Standard ML* and compiled with *MLton*'s whole program optimizations [11]) does the job nearly five times faster while using only 2720 megabytes of memory.

8 Related Work and Conclusions

Our term representation is inspired by intuitionistic implicational linear logic in natural deduction style which has explicit operations for weakening and contraction [5]. With explicit weakening and contraction, one easily maintains complete information about the free variables of a term at each node [15]. Our

term representation incorporates weakening and contraction into lambda abstraction. By using inspiration from ordered logic, we reduce the stored information at application nodes to a minimum, namely an integer; further, our variable nodes need to carry no information at all.

Another means to maintain information about free variables are *director strings* by Sinot [24]. Application nodes come with a map that tell for each variable whether it appears in the left or the right subterm or in both. Our term representation can be seen as an optimized version of director strings, however, we have no experimental comparison. Sinot et. al. [10] present some performance results of director strings; however, it is restricted to evaluation of some specific big lambda-terms. There is no study on their relative performance in a realistic application—yet that is our concern.

An alternative to explicit substitutions is Nadathur's suspension calculus [19], which, in a refinement, also maintains information about closedness of subterms. In this refinement, the suspension calculus maintains at least partial information about the free variables of a subterm. As the basis of an implementation of λ-Prolog [20], Nadathur has proven the efficiency of his term representation not only for normalization and equality checking, but also for higher-order unification.

Building on the suspension calculus, Liang, Nadathur, and Qi [17] have evaluated different term representations in the context of λProlog, a study that compares to our study of term representations for the Edinburgh Logical Framework. They have tested different combinations of features, confirming our result that lazy substitution is preferable to eager substitution [Beta Normal Forms], even more so when several substitutions are gathered into one traversal [Closures, Ordered]. They also test a variant where each term is equipped with an *annotation*, a flag telling whether this term is open, i. e., has free variables, or closed, i. e., has no free variables. In their experimental evaluation, these annotations pay off greatly for the poorly behaving eager substitution, yet give negligible advantage for explicit substitutions. It is hypothesized that in a combined substitution, each subterm will mention at least one variable with a high probability, so the traversal has to run over most of the whole term—this is certainly different in the substitution for a single variable.

To summarize, we have presented a new term representation for the lambda-calculus inspired by ordered linear logic, and experimentally compared it with well-known representations (closures, normal forms) in a prototypical implementation of a type checker for the Edinburgh Logical Framework. The experiments were carried out on large realistic proof terms, constructed manually and mechanically.

The results were not significantly in favor of our new representation. This might be due to the application domain, LF signature checking. For one, LF-definitions are closed, which means that substitutions never need to traverse a definition body when the definition is expanded, and this optimization is shared by all the term representations we compared. Secondly, we only tested type checking, not type reconstruction via unification. During type checking, where equality tests are expected to succeed, full normal forms are always computed, and closures are very short-lived in memory. More space leaks are to be expected in applications such as logic programming or type reconstruction, where unification is needed, which is not expected to always succeed. In constraint-based unification, unsolvable constraints might be postponed, keeping closures alive for longer. In such situations, the benefits of our representation might be more noticeable, more experiments are required.

In the future, we plan to investigate further term representations such as term graphs, and perform more experiments. The literature on experimentally successful term representations is sparse, our work contributes to close this gap. Our long term goal is to find a term representation which speeds up Agda's type reconstruction.

Acknowledgements The idea for the here presented ordered term representation was planted in a discussion with Christophe Raffalli who invited the first author to Chambery in February 2010. He mentioned to me that in his library `bindlib` formation of closures restricts the environment to the variables actually occurring free in the code. I also benefited from discussions with Brigitte Pientka and Stefan Monnier.

Thanks to Gabriel Scherer for comments on a draft version of this paper and pointers to related work. The final version of this paper was stylistically improved with the highly appreciated help of Neil Sculthorpe.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien & Jean-Jacques Lévy (1991): *Explicit Substitutions*. *Journal of Functional Programming* 1(4), pp. 375–416, doi:10.1017/S0956796800000186.

[2] Stephen Adams (1993): *Efficient Sets - A Balancing Act*. *Journal of Functional Programming* 3(4), pp. 553–561, doi:10.1017/S0956796800000885.

[3] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan & Daniel C. Wang (2010): *Semantic foundations for typed assembly languages*. *ACM Transactions on Programming Languages and Systems* 32(3), doi:10.1145/1709093.1709094.

[4] Henk Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*. North Holland, Amsterdam.

[5] Nick Benton, Gavin M. Bierman, Valeria de Paiva & Martin Hyland (1993): *A Term Calculus for Intuitionistic Linear Logic*. In Marc Bezem & Jan Friso Groote, editors: *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, *Lecture Notes in Computer Science* 664, Springer-Verlag, pp. 75–90, doi:10.1007/BFb0037099.

[6] Mathieu Boespflug (2010): *Conversion by Evaluation*. In Manuel Carro & Ricardo Peña, editors: *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, *Lecture Notes in Computer Science* 5937, Springer-Verlag, pp. 58–72, doi:10.1007/978-3-642-11503-5_7.

[7] N. G. de Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. *Indagationes Mathematicae* 34, pp. 381–392, doi:10.1016/1385-7258(72)90034-0.

[8] Iliano Cervesato & Frank Pfenning (2003): *A Linear Spine Calculus*. *Journal of Logic and Computation* 13(5), pp. 639–688, doi:10.1093/logcom/13.5.639.

[9] Thierry Coquand (1996): *An Algorithm for Type-Checking Dependent Types*. In: *Mathematics of Program Construction. Selected Papers from the Third International Conference on the Mathematics of Program Construction (July 17–21, 1995, Kloster Irsee, Germany)*, *Science of Computer Programming* 26, Elsevier, pp. 167–177, doi:10.1016/0167-6423(95)00021-6.

[10] Maribel Fernández, Ian Mackie & François-Régis Sinot (2005): *Lambda-Calculus with Director Strings*. *Applicable Algebra in Engineering, Communication and Computing* 15(6), pp. 393–437, doi:10.1007/s00200-005-0169-9.

[11] Matthew Fluet & Stephen Weeks (2001): *Contification Using Dominators*. In: *Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming (ICFP 2001)*, pp. 2–13, doi:10.1145/507635.507639.

[12] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

[13] Benjamin Grégoire & Xavier Leroy (2002): *A compiled implementation of strong reduction*. In: *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*,

Pittsburgh, Pennsylvania, USA, October 4-6, 2002, *SIGPLAN Notices* 37, ACM Press, pp. 235–246, doi:10.1145/581478.581501.

[14] INRIA (2010): *The Coq Proof Assistant Reference Manual*, version 8.3 edition. INRIA. Available at `http://coq.inria.fr/`.

[15] Delia Kesner & Stéphane Lengrand (2007): *Resource operators for lambda-calculus*. *Information and Computation* 205(4), pp. 419–473, doi:10.1016/j.ic.2006.08.008.

[16] Nicolai Kraus (2011): *A Lambda Term Representation Based on Linear Ordered Logic*. Bachelor's thesis, Department of Computer Science, Ludwig-Maximilians-University Munich.

[17] Chuck Liang, Gopalan Nadathur & Xiaochu Qi (2005): *Choices in representation and reduction strategies for lambda terms in intensional contexts*. *Journal of Automated Reasoning* 33(2), pp. 89–132, doi:10.1007/s10817-004-6885-1.

[18] Conor McBride & James McKinna (2004): *The view from the left*. *Journal of Functional Programming* 14(1), pp. 69–111, doi:10.1017/S0956796803004829.

[19] Gopalan Nadathur (1999): *A Fine-Grained Notation for Lambda Terms and Its Use in Intensional Operations*. *Journal of Functional and Logic Programming* 1999(2). Available at `http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/1999/A99-02/A99-02.html`.

[20] Gopalan Nadathur (2001): *The Metalanguage lambda-Prolog and Its Implementation*. In: *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, pp. 1–20, doi:10.1007/3-540-44716-4_1.

[21] Ulf Norell (2007): *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden.

[22] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf - A Meta-Logical Framework for Deductive Systems*. In Harald Ganzinger, editor: *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, *Lecture Notes in Computer Science* 1632, Springer-Verlag, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[23] Jeff Polakow & Frank Pfenning (1999): *Natural Deduction for Intuitionistic Non-communicative Linear Logic*. In Jean-Yves Girard, editor: *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, *Lecture Notes in Computer Science* 1581, Springer-Verlag, pp. 295–309, doi:10.1007/3-540-48959-2_21.

[24] François-Régis Sinot (2005): *Director Strings Revisited: A Generic Approach to the Efficient Representation of Free Variables in Higher-order Rewriting*. *Journal of Logic and Computation* 15(2), pp. 201–218, doi:10.1093/logcom/exi010.

[25] Kevin Watkins, Iliano Cervesato, Frank Pfenning & David Walker (2003): *A concurrent logical framework I: Judgements and properties*. Technical Report, School of Computer Science, Carnegie Mellon University, Pittsburgh.