2-level λ-lifting

Flemming Nielson [*] Hanne R. Nielson [†]

Abstract

The process of λ-lifting (or bracket abstraction) translates expressions in a typed λ-calculus into expressions in a typed combinator language. This is of interest because it shows that the λ-calculus and the combinator language are equally expressive (as the translation from combinators to λ-expressions is rather trivial). This paper studies the similar problems for 2-level λ-calculi and 2-level combinator languages. The 2-level nature of the type system enforces a formal distinction between binding times, e.g. between computations at compile-time and computations at run-time. In this setting the natural formulations of 2-level λ-calculi and 2-level combinator languages turn out *not* to be equally expressive. The translation into 2-level λ-calculus is straight-forward but the 2-level λ-calculus is too powerful for λ-lifting to succeed. We then develop a restriction of the 2-level λ-calculus for which λ-lifting succeeds and that is as expressive as the 2-level combinator language.

1 Introduction

Modern functional languages are often built as enrichments of the λ-calculus. In the implementation of these languages various forms of combinators are useful, e.g. [15,16,3]. The success of this is due to the process of λ-lifting (or bracket abstraction) that allows to eliminate variables of a λ-expression thereby turning it into a combinator expression. The techniques used build on results developed by [13,2] and a recent exposition of the ideas may be found in [1]. The approach is equally applicable to typed and untyped languages and in this paper we shall only study the typed case. Following [1] we shall consider a type system with types t given by

$$t ::= A_i \mid t \times t \mid t \rightarrow t$$

where the A_i (for $i \in I$) are certain base types, $t \times t$ is the product type and $t \rightarrow t$ is the function type. It is possible also to add sum types and recursive types but for lack of space we shall not do so.

The distinction between compile-time and run-time is important for the efficient implementation of programming languages. In our previous work [10,11] we have made this distinction explicit by imposing a 2-level structure on the typed λ-calculus. The types tt will then be given by

$$tt ::= \overline{A_i} \mid \underline{A_i} \mid tt \;\overline{\times}\; tt \mid tt \;\underline{\times}\; tt \mid tt \;\overline{\rightarrow}\; tt \mid tt \;\underline{\rightarrow}\; tt$$

The essential intuition will be that objects of type $tt \;\overline{\rightarrow}\; tt$ are to be evaluated at compile-time whereas objects of type $tt \;\underline{\rightarrow}\; tt$ are to be evaluated at run-time. So from the point of view of the compiler it must perform the computations of type $tt \;\overline{\rightarrow}\; tt$ and generate code for those of type $tt \;\underline{\rightarrow}\; tt$.

A similar distinction is made for the expressions by having two copies of the λ-notation. So we shall e.g. have a λ-abstraction $\overline{\lambda} x_i[tt]\ldots$ for building functions of type $tt \;\overline{\rightarrow}\; tt'$ and another

[*]Department of Computer Science, The Technical University of Denmark, DK-2800 Lyngby, Denmark.
[†]Department of Mathematics and Computer Science, AUC, Strandvejen 19, DK-9000 Aalborg, Denmark.

λ-abstraction $\underline{\lambda} x_i[tt]\ldots$ for building functions of type $tt \underline{\rightarrow} tt'$. The idea is here that we want the compiler to generate code for the functions specified by $\underline{\lambda}$-abstractions whereas those specified by $\overline{\lambda}$-abstractions should be interpreted at compile-time. Thus we shall be interested in transforming the λ-calculus specifying run-time computations into combinator form while leaving the λ-calculus for the compile-time computations untouched. This process will be called 2-level λ-lifting (as we do not want to distinguish inherently between combinators and supercombinators [4]).

Let us illustrate the approach by an example. In the λ-calculus the function select returning the n'th element of a list l may be defined by

$$\mathrm{select} \equiv \mathrm{fix}\ (\lambda S.\ \lambda n.\ \lambda l.\ (=\cdot\ 1\cdot\ n)\rightarrow (\mathrm{hd}\cdot\ l),\ (S\cdot\ (-\cdot\ n\cdot\ 1)\cdot\ (\mathrm{tl}\cdot\ l)))$$

Assume now that the first parameter always will be known at compile-time and that the second will not be known until run-time. In [11] we give an algorithm that will transform select into

$$\mathrm{select'} \equiv \overline{\mathrm{fix}}(\overline{\lambda} S.\ \overline{\lambda} n.\ \underline{\lambda} l.(=\bar{\cdot}\ 1\bar{\cdot}\ n)\overline{\Rightarrow} (\mathrm{hd}\underline{\cdot}\ l),(S\bar{\cdot}\ (-\bar{\cdot}\ n\bar{\cdot}\ 1)\underline{\cdot}\ (\mathrm{tl}\underline{\cdot}\ l)))$$

where again overlining indicates that the computations are performed at compile-time and underlining that they are performed at run-time. The purpose of the 2-level λ-lifting will be to get rid of the variable l but to keep S and n since they will be bound at compile-time. So we shall aim at an expression like

$$\mathrm{select''} \equiv \overline{\mathrm{fix}}(\overline{\lambda} S.\ \overline{\lambda} n.\ (=\bar{\cdot}\ 1\bar{\cdot}\ n)\overline{\Rightarrow}\ \mathrm{hd},(S\bar{\cdot}\ (-\bar{\cdot}\ n\bar{\cdot}\ 1))\,\Box\ \mathrm{tl})$$

where $\Box$ denotes functional composition at the run-time level.

In our previous work we have studied the efficient implementation of two-level functional languages where the compile-time actions are expressed in λ-notation and the run-time actions in combinator notation. In [8] we show how to generate code for abstract machines based on the von Neumann architecture and in [6,9] we study the application of data flow analyses within the framework of abstract interpretation. The present paper can therefore be seen as filling in the gap between these results and the techniques of [10,11] for imposing a 2-level structure on λ-expressions. Finally, the algorithms presented in this paper have been implemented in a test bed system designed to experiment with various ideas related to 2-level functional languages.

2 Review of the 1-level case

As an introduction to our approach to 2-level λ-lifting we first review the usual concept of λ-lifting (or bracket abstraction). This does not add much to the explanations given in [1] but allows us to fix our notation by means of the familiar case.

First we define the typed λ-calculus DML_e that has the types

$$t ::= A_i \mid t \times t \mid t \rightarrow t$$

and expressions

$$e ::= f_i[t] \mid (e,e) \mid e \downarrow j \mid \lambda x_i[t].e \mid e \cdot e \mid x_i \mid \mathrm{fix}\ e \mid e \rightarrow e\ ,\ e$$

Here the $f_i[t]$ (for $i \in I$) are constants of the type indicated. Next we have pairing, projection, λ-abstraction, application, variable, fixed point and conditional. As we are in a typed language these expressions are subject to certain well-formedness conditions. The well-formedness predicate has the form $\mathrm{tenv} \vdash e{:}t$ where tenv is a type environment (i.e. a map from a finite set of variables to types) and says that e has type t. It is defined by

$$\mathrm{tenv} \vdash f_i[t]{:}t$$

$$\frac{\mathrm{tenv} \vdash e_1{:}t_1,\ \mathrm{tenv} \vdash e_2{:}t_2}{\mathrm{tenv} \vdash (e_1,e_2){:}t_1 \times t_2}$$

$$\frac{\text{tenv} \vdash e{:}t_1 \times t_2}{\text{tenv} \vdash e \downarrow j{:}t_j} \qquad \text{if } j = 1,2$$

$$\frac{\text{tenv}[x_i \mapsto t] \vdash e{:}t'}{\text{tenv} \vdash \lambda x_i[t].e{:}t \rightarrow t'}$$

$$\frac{\text{tenv} \vdash e_1{:}t' \rightarrow t,\ \text{tenv} \vdash e_2{:}t'}{\text{tenv} \vdash e_1 \cdot e_2{:}t}$$

$$\text{tenv} \vdash x_i{:}t \qquad \text{if tenv}(x_i) = t$$

$$\frac{\text{tenv} \vdash e{:}t \rightarrow t}{\text{tenv} \vdash \text{fix } e{:}t}$$

$$\frac{\text{tenv} \vdash e{:}A_{\text{bool}},\ \text{tenv} \vdash e_1{:}t,\ \text{tenv} \vdash e_2{:}t}{\text{tenv} \vdash e \rightarrow e_1,e_2{:}t}$$

Fact 1 Expressions are uniquely typed, i.e. if tenv $\vdash$ e:t_1 and tenv $\vdash$ e:t_2 then $t_1 = t_2$. □

The proof is by induction on the inference of tenv $\vdash$ e:t_1 and $t_1 = t_2$ means that the types are syntactically equal.

In a similar way we define the typed combinator language DML_m. It has types

$$t ::= A_i \mid t \times t \mid t \rightarrow t$$

and expressions

$$e ::= f_i[t] \mid \text{tuple}(e,e) \mid \text{take}_j[t] \mid \text{curry } e \mid \text{apply}[t] \mid e \,\square\, e \mid \text{fix}[t] \mid \text{cond}(e,e,e) \mid \text{const}[t]\ e \mid \text{id}[t]$$

Here tuple and take$_j$ relate to the product type and the intention is that tuple(f,g)(v) is (f(v),g(v)) and take$_j$[t](v_1,v_2) is v_j. For the function space we have curry and apply and here curry(f)(u)(v) is f(u,v) and apply[t](f,v) is f(v). Function composition is denoted by □ , fix[t] is the fixed point operator and the intended meaning of the conditional is that cond(f,g,h)(v) is g(v) if f(v) holds and otherwise h(v). Finally const[t] ignores one of its arguments so const[t](f)(v) is f and id[t] is the identity function.

The well-formedness predicate has the form $\vdash$ e:t and is defined by

$$\vdash f_i[t]{:}t$$

$$\frac{\vdash e_1{:}t \rightarrow t_1,\ \vdash e_2{:}t \rightarrow t_2}{\vdash \text{tuple}(e_1,e_2){:}t \rightarrow (t_1 \times t_2)}$$

$$\vdash \text{take}_j[t]{:}t \rightarrow t_j \qquad \text{if } t = t_1 \times t_2 \text{ and } j{=}1,2$$

$$\frac{\vdash e{:}(t_1 \times t_2) \rightarrow t_3}{\vdash \text{curry } e{:}t_1 \rightarrow (t_2 \rightarrow t_3)}$$

$$\vdash \text{apply}[t]{:}((t_1 \rightarrow t_2) \times t_1) \rightarrow t_2 \qquad \text{if } t = t_1 \rightarrow t_2$$

$$\frac{\vdash e_1{:}t_2 \rightarrow t_3,\ \vdash e_2{:}t_1 \rightarrow t_2}{\vdash e_1 \,\square\, e_2{:}t_1 \rightarrow t_3}$$

$$\vdash \text{fix}[t]{:}(t \rightarrow t) \rightarrow t$$

$$\frac{\vdash e_1{:}t \rightarrow A_{\text{bool}},\ \vdash e_2{:}t \rightarrow t',\ \vdash e_3{:}t \rightarrow t'}{\vdash \text{cond}(e_1,e_2,e_3){:}t \rightarrow t'}$$

$$\frac{\vdash e{:}t'}{\vdash \text{const}[t]\ e{:}t \rightarrow t'}$$

$$\vdash \text{id}[t]{:}t \rightarrow t$$

We have added sufficient type information to the combinators that we have the following analogue of Fact 1:

Fact 2 Expressions are uniquely typed, i.e. if $\vdash e{:}t_1$ and $\vdash e{:}t_2$ then $t_1 = t_2$. $\Box$

From a pragmatic point of view one might consider to constrain the type t of a constant $f_i[t]$ to be of the form $t_1 \rightarrow t_2$. We shall not do so as for the subsequent development to make sense we would either have to impose a similar constraint on constants of DML_e or else we should change the functionality of constants in the transformation to follow.

We now turn to the relationship between DML_e and DML_m. The transformation from DML_m to DML_e amounts to the expansion of the combinators into λ-expressions. A minor complication is that not all the necessary type information is explicitly present and we shall rely on Fact 2 in order to obtain it. We therefore formulate the process as the definition of a function

$$\varepsilon\colon \{\ e \in DML_m \mid \exists t.\ \vdash e{:}t\ \} \rightarrow \{\ e \mid e \in DML_e\ \}$$

by means of the following equations which merely restate the intuitions about the combinators tuple, $take_j[t]$ etc. in a formal way:

$$\varepsilon[\![f_i[t]]\!]= f_i[t]$$

$$\varepsilon[\![tuple(e_1,\ e_2)]\!]= \lambda x_a[t].\ (\varepsilon[\![e_1]\!]\cdot\ x_a,\ \varepsilon[\![e_2]\!]\cdot\ x_a) \quad \text{where} \vdash e_1{:}t \rightarrow t_1$$

$$\varepsilon[\![take_j[t]]\!]= \lambda x_a[t].\ x_a \downarrow j$$

$$\varepsilon[\![curry\ e]\!]= \lambda x_a[t_1].\ \lambda x_b[t_2].\ \varepsilon[\![e]\!]\cdot\ (x_a,\ x_b) \quad \text{where} \vdash e\colon (t_1 \times t_2) \rightarrow t$$

$$\varepsilon[\![apply\ [t]]\!]= \lambda x_a[(t_1{\rightarrow}t_2){\times}t_1].\ (x_a \downarrow 1)\cdot\ (x_a \downarrow 2) \quad \text{where } t = t_1 \rightarrow t_2$$

$$\varepsilon[\![e_1 \Box e_2]\!]= \lambda x_a[t].\ \varepsilon[\![e_1]\!]\cdot\ (\varepsilon[\![e_2]\!]\cdot\ x_a) \quad \text{where} \vdash e_2{:}t \rightarrow t'$$

$$\varepsilon[\![fix[t]]\!]= \lambda x_a[t \rightarrow t].\ fix\ x_a$$

$$\varepsilon[\![cond(e_1,\ e_2,\ e_3)]\!]= \lambda x_a[t].\ \varepsilon[\![e_1]\!]\cdot\ x_a \rightarrow \varepsilon[\![e_2]\!]\cdot\ x_a,\ \varepsilon[\![e_3]\!]\cdot\ x_a \quad \text{where} \vdash e_1\colon\ t \rightarrow A_{bool}$$

$$\varepsilon[\![const[t]\ e]\!]= \lambda x_a[t].\ \varepsilon[\![e]\!]$$

$$\varepsilon[\![id[t]]\!]= \lambda x_a[t].\ x_a$$

To see that this is a correct translation we note that

Fact 3 The transformation ε preserves the types of expressions, i.e. if $\vdash e{:}t$ then $\emptyset \vdash \varepsilon[\![e]\!]{:}t$. $\Box$

where $\emptyset$ denotes the empty type environment. Hopefully it is intuitively clear that it also preserves the semantics. If we were to be formal about this we could define reduction rules for DML_e and DML_m and use this as a basis for relating the semantics (see [1]). Alternatively, we could define a denotational semantics with a suitable notion of interpretation of the primitives (along the lines of [8,9]). However, we shall not pursue this further here.

Concerning the translation from DML_e to DML_m we consider an expression e of DML_e that has type t, i.e. that satisfies tenv $\vdash e{:}t$. Assuming that tenv has a nonempty domain $\{x_1,\ \cdots,\ x_n\}$ and maps x_i to t_i the type of the translated term will be of the form $(\cdots(t_1{\times}t_2){\times}t_3 \cdots{\times}t_n) \rightarrow t$. To make this precise we shall let a *position environment* penv be a list of pairs of variables and types. The underlying type environment then is

$$\rho(penv) = \lambda x_i. \begin{cases} \text{undefined} & \text{if no } penv{\downarrow}j{\downarrow}1 \text{ is } x_i \\ penv{\downarrow}j{\downarrow}2 & \text{if j is minimal s.t. } penv{\downarrow}j{\downarrow}1 \text{ is } x_i \end{cases}$$

and the product of the variable types is

$$\Pi(penv) = \begin{cases} \text{undefined} & \text{if } penv = () \\ t & \text{if } penv = ((x,t)) \\ \Pi(penv') \times t & \text{if } penv = ((x,t))\hat{\ }penv' \end{cases}$$

So if penv is $((x_1,A_1)(x_2,A_2)(x_3,A_3))$ then $\rho(\text{penv})$ maps x_i to A_i and $\Pi(\text{penv})$ is $(A_3 \times A_2)\times A_1$. The intention is that the transformed version $\Lambda^{\text{penv}}[\![e]\!]$ has type $\Pi(\text{penv}) \rightarrow t$ whenever $\rho(\text{penv}) \vdash e{:}t$. We shall not allow the case where penv = () and so if $\emptyset \vdash e{:}t$ we must artificially add a dummy variable and a dummy type. (This is in line with [1] but we shall need to be more careful when we come to 2-level λ-lifting!) To assist in the definition of Λ^{penv} we need the function

$$\pi_j^{\text{penv}} = \begin{cases} \text{undefined} & \text{if no penv}\downarrow j\downarrow 1 \text{ is } x_j \\ \text{id}[\Pi(\text{penv})] & \text{if penv} = ((x_j,t)) \\ \text{take}_2[\Pi(\text{penv})] & \text{if penv} = ((x_j,t))\hat{\ }\text{penv}' \\ \pi_j^{\text{penv}'} \,\square\, \text{take}_1[\Pi(\text{penv})] & \text{if penv} = ((x_i,t))\hat{\ }\text{penv}' \text{ and } i \neq j \end{cases}$$

for locating the component in $\Pi(\text{penv})$ that corresponds to x_j. For the example above we have

$$\pi_1^{\text{penv}} = \text{take}_2[(A_3 \times A_2)\times A_1],$$
$$\pi_2^{\text{penv}} = \text{take}_2[A_3 \times A_2] \,\square\, \text{take}_1[(A_3 \times A_2)\times A_1]$$
$$\pi_3^{\text{penv}} = \text{id}[A_3] \,\square\, \text{take}_1[A_3 \times A_2] \,\square\, \text{take}_1[(A_3 \times A_2)\times A_1]$$

In analogy with the definition of ε we shall use Fact 1 to define a function

$$\Lambda^{\text{penv}}\colon \{ e \in \text{DML}_e \mid \exists t.\ \rho(\text{penv}) \vdash e{:}t \} \rightarrow \{ e \mid e \in \text{DML}_m \}$$

whenever penv $\neq$ () by

$$\Lambda^{\text{penv}}[\![f_i[t]]\!] = \text{const}[\Pi(\text{penv})]\ f_i[t]$$
$$\Lambda^{\text{penv}}[\![(e_1, e_2)]\!] = \text{tuple}(\Lambda^{\text{penv}}[\![e_1]\!], \Lambda^{\text{penv}}[\![e_2]\!])$$
$$\Lambda^{\text{penv}}[\![e\downarrow j]\!] = \text{take}_j[t] \,\square\, \Lambda^{\text{penv}}[\![e]\!] \quad \text{where } \rho(\text{penv}) \vdash e{:}t$$
$$\Lambda^{\text{penv}}[\![\lambda x_i[t].e]\!] = \text{curry}\ \Lambda^{((x_i,t))\hat{\ }\text{penv}}[\![e]\!]$$
$$\Lambda^{\text{penv}}[\![e_1 \cdot e_2]\!] = \text{apply}\ [t_1 \rightarrow t_2] \,\square\, \text{tuple}(\Lambda^{\text{penv}}[\![e_1]\!], \Lambda^{\text{penv}}[\![e_2]\!]) \qquad \text{where } \rho(\text{penv}) \vdash e_1{:}t_1 \rightarrow t_2$$
$$\Lambda^{\text{penv}}[\![x_i]\!] = \pi_i^{\text{penv}}$$
$$\Lambda^{\text{penv}}[\![\text{fix } e]\!] = \text{fix}[t] \,\square\, \Lambda^{\text{penv}}[\![e]\!] \quad \text{where } \rho(\text{penv}) \vdash e{:}t \rightarrow t$$
$$\Lambda^{\text{penv}}[\![e_1 \rightarrow e_2, e_3]\!] = \text{cond}(\Lambda^{\text{penv}}[\![e_1]\!], \Lambda^{\text{penv}}[\![e_2]\!], \Lambda^{\text{penv}}[\![e_3]\!])$$

That this is a well-behaved definition that lives up to the claims is expressed by

Fact 4 If penv $\neq$ () and $\rho(\text{penv}) \vdash e{:}t$ then $\vdash \Lambda^{\text{penv}}[\![e]\!]{:}\Pi(\text{penv}) \rightarrow t$. $\square$

Hopefully, it is also intuitively clear that Λ^{penv} preserves the semantics and as above we shall not be more formal about this. Because of the lack of space we must refer to any standard textbook, e.g. [1], for examples of the translation.

3 2-level λ-calculi and combinator languages

After the above review we can now approach 2-level λ-lifting. In the 2-level notations we replace the type system of the previous section with

$$tt ::= \overline{A_i} \mid \underline{A_i} \mid tt \mathbin{\overline{\times}} tt \mid tt \mathbin{\underline{\times}} tt \mid tt \mathbin{\overline{\rightarrow}} tt \mid tt \mathbin{\underline{\rightarrow}} tt$$

as was already mentioned in the Introduction. Here overlining is used to indicate early binding and our prime example of this is compile-time and similarly underlining is used to indicate late binding and here the prime example is run-time. The considerations of compile-time versus run-time motivate defining the following well-formedness predicate $\vdash tt{:}k$ for when a type tt is well-formed of kind $k \in \{c,r\}$. Clearly c will correspond to compile-time and r to run-time. The definition is

tt	$\vdash$ tt:c	$\vdash$ tt:r
$\overline{A}_i$	true	false
$\underline{A}_i$	false	true
$tt_1 \mathbin{\overline{\times}} tt_2$	$\vdash tt_1{:}c \wedge \vdash tt_2{:}c$	false
$tt_1 \mathbin{\underline{\times}} tt_2$	false	$\vdash tt_1{:}r \wedge \vdash tt_2{:}r$
$tt_1 \mathbin{\overline{\rightarrow}} tt_2$	$\vdash tt_1{:}c \wedge \vdash tt_2{:}c$	false
$tt_1 \mathbin{\underline{\rightarrow}} tt_2$	$\vdash tt_1{:}r \wedge \vdash tt_2{:}r$	$\vdash tt_1{:}r \wedge \vdash tt_2{:}r$

Here no compile-time types can be embedded in run-time types; this is motivated by the fact that compile-time takes place before run-time. A run-time type of the form $tt_1 \mathbin{\underline{\rightarrow}} tt_2$ is also a compile-time type; this is motivated by the fact that a compiler may manipulate code (and code corresponds to run-time computations) but not the actual values that arise at run-time. One may of course consider variations in this definition but the present definition has been found useful for abstract interpretation and code generation [6,8].

The idea with the 2-level notations is that we have a choice of using λ-expressions or combinators at the compile-time level and a similar choice at the run-time level. This gives a total of four languages but we shall restrict ourselves to the case where we always use λ-expressions at the compile-time level. The notation where we use λ-expressions at both levels will be called TML_e and has expressions given by

$$\begin{array}{rl} te ::= & f_i[tt] \mid x_i \mid \overline{(te,\ te)} \mid te \mathbin{\overline{\downarrow}} j \mid \overline{\lambda} x_i[tt].te \mid te \mathbin{\overline{\cdot}} te \mid \overline{\mathrm{fix}}\ te \mid te \mathbin{\overline{\rightarrow}} te,\ te \\ & \mid \underline{(te,\ te)} \mid te \mathbin{\underline{\downarrow}} j \mid \underline{\lambda} x_i[tt].te \mid te \mathbin{\underline{\cdot}} te \mid \underline{\mathrm{fix}}\ te \mid te \mathbin{\underline{\rightarrow}} te,\ te \end{array}$$

Again overlining is used for the compile-time level and underlining is used for the run-time level. For the well-formedness predicate we propose the following generalization of the one for DML_e. The form of the predicate is tenv $\vdash$ te:tt and the definition is

$$\mathrm{tenv} \vdash f_i[tt]{:}tt \qquad \text{if } \exists k.\ \vdash tt{:}k$$

$$\mathrm{tenv} \vdash x_i{:}tt \qquad \text{if } \mathrm{tenv}(x_i) = tt \text{ and } \exists k.\ \vdash tt{:}k$$

$$\frac{\mathrm{tenv} \vdash te_1{:}tt_1,\ \mathrm{tenv} \vdash te_2{:}tt_2}{\mathrm{tenv} \vdash \overline{(te_1,\ te_2)}{:}tt_1 \mathbin{\overline{\times}} tt_2} \qquad \text{if} \vdash tt_1{:}c \text{ and} \vdash tt_2{:}c$$

$$\frac{\mathrm{tenv} \vdash te{:}tt_1 \mathbin{\overline{\times}} tt_2}{\mathrm{tenv} \vdash te \mathbin{\overline{\downarrow}} j{:}tt_j} \qquad \text{if } j = 1,2$$

$$\frac{\mathrm{tenv}[x_i \mapsto tt] \vdash te{:}tt'}{\mathrm{tenv} \vdash \overline{\lambda} x_i[tt].te{:}tt \mathbin{\overline{\rightarrow}} tt'} \qquad \text{if} \vdash tt{:}c \text{ and} \vdash tt'{:}c$$

$$\frac{\mathrm{tenv} \vdash te_1{:}tt' \mathbin{\overline{\rightarrow}} tt,\ \mathrm{tenv} \vdash te_2{:}tt'}{\mathrm{tenv} \vdash te_1 \mathbin{\overline{\cdot}} te_2{:}tt}$$

$$\frac{\mathrm{tenv} \vdash te{:}tt \mathbin{\overline{\rightarrow}} tt}{\mathrm{tenv} \vdash \overline{\mathrm{fix}}\ te{:}tt}$$

$$\frac{\mathrm{tenv} \vdash te{:}\overline{A}_{\mathrm{bool}},\ \mathrm{tenv} \vdash te_1{:}tt,\ \mathrm{tenv} \vdash te_2{:}tt}{\mathrm{tenv} \vdash te \mathbin{\overline{\rightarrow}} te_1,\ te_2{:}tt}$$

$$\frac{\mathrm{tenv} \vdash te_1{:}tt_1,\ \mathrm{tenv} \vdash te_2{:}tt_2}{\mathrm{tenv} \vdash \underline{(te_1,\ te_2)}{:}tt_1 \mathbin{\underline{\times}} tt_2} \qquad \text{if} \vdash tt_1{:}r \text{ and} \vdash tt_2{:}r$$

$$\frac{\mathrm{tenv} \vdash te{:}tt_1 \mathbin{\underline{\times}} tt_2}{\mathrm{tenv} \vdash te \mathbin{\underline{\downarrow}} j{:}tt_j} \qquad \text{if } j = 1,2$$

$$\frac{\mathrm{tenv}[x_i \mapsto \mathrm{tt}] \vdash \mathrm{te}:\mathrm{tt}'}{\mathrm{tenv} \vdash \underline{\lambda} x_i[\mathrm{tt}].\mathrm{te}:\mathrm{tt}\underline{\rightarrow}\mathrm{tt}'} \quad \text{if } \vdash \mathrm{tt}:\mathrm{r} \text{ and } \vdash \mathrm{tt}':\mathrm{r}$$

$$\frac{\mathrm{tenv} \vdash \mathrm{te}_1:\mathrm{tt}'\underline{\rightarrow}\mathrm{tt},\ \mathrm{tenv} \vdash \mathrm{te}_2:\mathrm{tt}'}{\mathrm{tenv} \vdash \mathrm{te}_1 \ \underline{\cdot}\ \mathrm{te}_2:\mathrm{tt}}$$

$$\frac{\mathrm{tenv} \vdash \mathrm{te}:\mathrm{tt}\underline{\rightarrow}\mathrm{tt}}{\mathrm{tenv} \vdash \underline{\mathrm{fix}}\ \mathrm{te}:\mathrm{tt}}$$

$$\frac{\mathrm{tenv} \vdash \mathrm{te}:\underline{\mathrm{A}}_{\mathrm{bool}},\ \mathrm{tenv} \vdash \mathrm{te}_1:\mathrm{tt},\ \mathrm{tenv} \vdash \mathrm{te}_2:\mathrm{tt}}{\mathrm{tenv} \vdash \mathrm{te} \ \underline{\rightarrow}\ \mathrm{te}_1,\ \mathrm{te}_2:\mathrm{tt}}$$

With respect to the rules for DML_e one may note that essentially we have two copies of these but we need to add additional side conditions of the form $\vdash$ tt:k in order for the constructed types to be well-formed. From an intuitive point of view it is unclear whether one should add the constraint that $\vdash$ tt:r in the rule for te $\underline{\rightarrow}$ te, te; however, in Section 4 we shall see a formal reason for imposing this constraint. We have

Fact 5 Expressions have well-formed types, i.e. if tenv $\vdash$ te:tt then $\exists$k. $\vdash$ tt:k. □

and in analogy with Fact 1 we also have

Fact 6 Expressions are uniquely typed, i.e. if tenv $\vdash$ te:tt_1 and tenv $\vdash$ te:tt_2 then $\mathrm{tt}_1 = \mathrm{tt}_2$. □

Furthermore we claim that TML_e is a natural analogue of DML_e but for the 2-level case. Clearly one can translate an expression in TML_e into one in DML_e (by removing all underlining and overlining) and it is shown in [10,11] that it is also possible to translate expressions in DML_e into TML_e.

Another 2-level notation is TML_m where we use combinators at the run-time level. The types tt are as above and so is the well-formedness predicate for types. For expressions the syntax is

$$\begin{aligned}\mathrm{te} ::= \mathrm{f}_i[\mathrm{tt}] \mid x_i &\mid \overline{(}\mathrm{te}, \mathrm{te}\overline{)} \mid \mathrm{te}\,\overline{\downarrow}\,\overline{\mathrm{j}} \mid \overline{\lambda} x_i[\mathrm{tt}].\mathrm{te} \mid \mathrm{te}\ \overline{\cdot}\ \mathrm{te} \mid \overline{\mathrm{fix}}\ \mathrm{te} \mid \mathrm{te} \overline{\Rightarrow} \mathrm{te}, \mathrm{te}\\ &\mid \underline{\mathrm{tuple}}(\mathrm{te},\mathrm{te}) \mid \underline{\mathrm{take}_j}[\mathrm{tt}] \mid \underline{\mathrm{curry}}\ \mathrm{te} \mid \underline{\mathrm{apply}}[\mathrm{tt}] \mid \mathrm{te} \,\square\, \mathrm{te}\\ &\mid \underline{\mathrm{fix}}\ \mathrm{te} \mid \underline{\mathrm{cond}}(\mathrm{te},\mathrm{te},\mathrm{te}) \mid \underline{\mathrm{const}}[\mathrm{tt}]\ \mathrm{te} \mid \underline{\mathrm{id}}[\mathrm{tt}]\end{aligned}$$

The well-formedness predicate has the form tenv $\vdash$ te:tt and is defined by

$$\left.\begin{array}{l}\mathrm{tenv} \vdash \mathrm{f}_i[\mathrm{tt}]:\mathrm{tt} \quad \text{if } \exists \mathrm{k}.\ \vdash \mathrm{tt}:\mathrm{k}\\ \quad\vdots\\ \dfrac{\mathrm{tenv} \vdash \mathrm{te}:\overline{\mathrm{A}}_{\mathrm{bool}},\ \mathrm{tenv} \vdash \mathrm{te}_1:\mathrm{tt},\ \mathrm{tenv} \vdash \mathrm{te}_2:\mathrm{tt}}{\mathrm{tenv} \vdash \mathrm{te} \overline{\Rightarrow} \mathrm{te}_1,\ \mathrm{te}_2:\mathrm{tt}}\end{array}\right\} \text{as above}$$

$$\frac{\mathrm{tenv} \vdash \mathrm{te}_1:\mathrm{tt}\underline{\rightarrow}\mathrm{tt}_1,\ \mathrm{tenv} \vdash \mathrm{te}_2:\mathrm{tt}\underline{\rightarrow}\mathrm{tt}_2}{\mathrm{tenv} \vdash \underline{\mathrm{tuple}}(\mathrm{te}_1,\mathrm{te}_2):\mathrm{tt}\underline{\rightarrow}(\mathrm{tt}_1\underline{\times}\mathrm{tt}_2)}$$

$$\mathrm{tenv} \vdash \underline{\mathrm{take}_j}[\mathrm{tt}]:\mathrm{tt}\underline{\rightarrow}\mathrm{tt}_j \quad \text{if } \vdash \mathrm{tt}:\mathrm{r} \text{ and } \mathrm{tt} = \mathrm{tt}_1\underline{\times}\mathrm{tt}_2 \text{ and } j=1,2$$

$$\frac{\mathrm{tenv} \vdash \mathrm{te}:(\mathrm{tt}_1\underline{\times}\mathrm{tt}_2)\underline{\rightarrow}\mathrm{tt}_3}{\mathrm{tenv} \vdash \underline{\mathrm{curry}}\ \mathrm{te}:\mathrm{tt}_1\underline{\rightarrow}(\mathrm{tt}_2\underline{\rightarrow}\mathrm{tt}_3)}$$

$$\mathrm{tenv} \vdash \underline{\mathrm{apply}}[\mathrm{tt}]:((\mathrm{tt}_1\underline{\rightarrow}\mathrm{tt}_2)\underline{\times}\mathrm{tt}_1)\underline{\rightarrow}\mathrm{tt}_2 \quad \text{if } \vdash \mathrm{tt}:\mathrm{r} \text{ and } \mathrm{tt} = \mathrm{tt}_1\underline{\rightarrow}\mathrm{tt}_2$$

$$\frac{\mathrm{tenv} \vdash \mathrm{te}_1:\mathrm{tt}_2\underline{\rightarrow}\mathrm{tt}_3,\ \mathrm{tenv} \vdash \mathrm{te}_2:\mathrm{tt}_1\underline{\rightarrow}\mathrm{tt}_2}{\mathrm{tenv} \vdash \mathrm{te}_1 \,\square\, \mathrm{te}_2:\mathrm{tt}_1\underline{\rightarrow}\mathrm{tt}_3}$$

$$\mathrm{tenv} \vdash \underline{\mathrm{fix}}[\mathrm{tt}]:(\mathrm{tt}\underline{\rightarrow}\mathrm{tt})\underline{\rightarrow}\mathrm{tt} \quad \text{if } \vdash \mathrm{tt}:\mathrm{r}$$

$$\frac{\mathrm{tenv} \vdash \mathrm{te}_1:\mathrm{tt}\underline{\rightarrow}\underline{\mathrm{A}}_{\mathrm{bool}},\ \mathrm{tenv} \vdash \mathrm{te}_2:\mathrm{tt}\underline{\rightarrow}\mathrm{tt}',\ \mathrm{tenv} \vdash \mathrm{te}_3:\mathrm{tt}\underline{\rightarrow}\mathrm{tt}'}{\mathrm{tenv} \vdash \underline{\mathrm{cond}}(\mathrm{te}_1,\mathrm{te}_2,\mathrm{te}_3):\mathrm{tt}\underline{\rightarrow}\mathrm{tt}'}$$

$$\frac{\text{tenv} \vdash \text{te:tt'}}{\text{tenv} \vdash \underline{\text{const}}[\text{tt}]\ \text{te:tt}\underline{\rightarrow}\text{tt'}} \quad \text{if} \vdash \text{tt:r and} \vdash \text{tt':r}$$

$$\text{tenv} \vdash \underline{\text{id}}[\text{tt}]\text{:tt}\underline{\rightarrow}\text{tt} \qquad \text{if} \vdash \text{tt:r}$$

The rules for well-formedness of the top-level terms are as in TML_e. In particular we do not constrain the types tt of the constants $\text{f}_\text{i}[\text{tt}]$ although it might seem unfit to have constants of run-time types that are not function types. (However, one may introduce the constraint $\vdash$ tt:c if desired and if also the type tt of a conditional $\text{te}_1 \overline{\Rightarrow} \text{te}_2,\text{te}_3$ is constrained in this way then TML_m would be a subset of the language considered in [8].) In analogy with the results for TML_e we have

Fact 7 Expressions have well-formed types, i.e. if tenv $\vdash$ te:tt then $\exists$k. $\vdash$ tt:k. □

Fact 8 Expressions are uniquely typed, i.e. if tenv $\vdash$ te:tt_1 and tenv $\vdash$ te:tt_2 then $\text{tt}_1 = \text{tt}_2$. □

As in the case of DML_e and DML_m the expansion of combinators is rather straight-forward. So we define a function

$$\varepsilon_{\text{tenv}}\text{: } \{ \text{ te} \in \text{TML}_m \mid \exists\text{tt. tenv} \vdash \text{te:tt } \} \rightarrow \{ \text{ te} \mid \text{ te} \in \text{TML}_e \ \}$$

for each type environment tenv. We need the type environment because we shall rely on Fact 8 in order to infer missing type information (just as we used Fact 2 in the 1-level case). The definition is

$$\varepsilon_{\text{tenv}}\ [[\text{f}_\text{i}[\text{tt}]]] = \text{f}_\text{i}[\text{tt}]$$

$$\varepsilon_{\text{tenv}}\ [[\text{x}_\text{i}]] = \text{x}_\text{i}$$

$$\varepsilon_{\text{tenv}}\ [[\overline{(}\text{te}_1,\ \text{te}_2\ \overline{)}]] = \overline{(}\varepsilon_{\text{tenv}}\ [[\text{te}_1]],\ \varepsilon_{\text{tenv}}\ [[\text{te}_2]]\overline{)}$$

$$\varepsilon_{\text{tenv}}\ [[\text{te}\overline{\downarrow \text{j}}]] = \varepsilon_{\text{tenv}}\ [[\text{te}]]\overline{\downarrow \text{j}}$$

$$\varepsilon_{\text{tenv}}\ [[\overline{\lambda}\text{x}_\text{i}[\text{tt}].\text{te}]] = \overline{\lambda}\text{x}_\text{i}[\text{tt}].\ \varepsilon_{\text{tenv}[\text{x}_\text{i}\mapsto\text{tt}]}\ [[\text{te}]]$$

$$\varepsilon_{\text{tenv}}\ [[\text{te}_1\ \overline{\cdot}\ \text{te}_2]] = \varepsilon_{\text{tenv}}\ [[\text{te}_1]]\overline{\cdot}\ \varepsilon_{\text{tenv}}\ [[\text{te}_2]]$$

$$\varepsilon_{\text{tenv}}\ [[\overline{\text{fix}}\ \text{te}]] = \overline{\text{fix}}\ \varepsilon_{\text{tenv}}\ [[\text{te}]]$$

$$\varepsilon_{\text{tenv}}\ [[\text{te}_1 \overline{\Rightarrow} \text{te}_2,\ \text{te}_3]] = \varepsilon_{\text{tenv}}\ [[\text{te}_1]] \overline{\Rightarrow} \varepsilon_{\text{tenv}}\ [[\text{te}_2]],\ \varepsilon_{\text{tenv}}\ [[\text{te}_3]]$$

$$\varepsilon_{\text{tenv}}\ [[\underline{\text{tuple}}(\text{te}_1,\text{te}_2)]] = \underline{\lambda}\text{x}_\text{a}[\text{tt}].\ \underline{(}\varepsilon_{\text{tenv}}\ [[\text{te}_1]]\underline{\cdot}\ \text{x}_\text{a},\ \varepsilon_{\text{tenv}}\ [[\text{te}_2]]\underline{\cdot}\ \text{x}_\text{a}\ \underline{)} \quad \text{where tenv} \vdash \text{te}_1\text{:tt}\underline{\rightarrow}\text{tt}_1$$

$$\varepsilon_{\text{tenv}}\ [[\underline{\text{take}}_\text{j}[\text{tt}]]] = \underline{\lambda}\text{x}_\text{a}[\text{tt}].\ \text{x}_\text{a}\ \underline{\downarrow \text{j}}$$

$$\varepsilon_{\text{tenv}}\ [[\underline{\text{curry}}\ \text{te}]] = \underline{\lambda}\text{x}_\text{a}[\text{tt}_1].\ \underline{\lambda}\text{x}_\text{b}[\text{tt}_2].\ \varepsilon_{\text{tenv}}\ [[\text{te}]]\underline{\cdot}\ \underline{(}\text{x}_\text{a},\ \text{x}_\text{b}\underline{)} \quad \text{where tenv} \vdash \text{te:}(\text{tt}_1\underline{\times}\text{tt}_2)\underline{\rightarrow}\text{tt}$$

$$\varepsilon_{\text{tenv}}\ [[\underline{\text{apply}}[\text{tt}]]] = \underline{\lambda}\text{x}_\text{a}[(\text{tt}_1\underline{\rightarrow}\text{tt}_2)\underline{\times}\text{tt}_1].\ (\text{x}_\text{a}\underline{\downarrow 1})\ \underline{\cdot}\ (\text{x}_\text{a}\underline{\downarrow 2}) \quad \text{where tt} = \text{tt}_1\underline{\rightarrow}\text{tt}_2$$

$$\varepsilon_{\text{tenv}}\ [[\text{te}_1\ \square\ \text{te}_2]] = \underline{\lambda}\text{x}_\text{a}[\text{tt}].\ \varepsilon_{\text{tenv}}\ [[\text{te}_1]]\underline{\cdot}\ (\varepsilon_{\text{tenv}}\ [[\text{te}_2]]\underline{\cdot}\ \text{x}_\text{a}) \quad \text{where tenv} \vdash \text{te}_2\text{:tt}\underline{\rightarrow}\text{tt'}$$

$$\varepsilon_{\text{tenv}}\ [[\underline{\text{fix}}[\text{tt}]]] = \underline{\lambda}\text{x}_\text{a}[\text{tt}\underline{\rightarrow}\text{tt}].\ \underline{\text{fix}}\ \text{x}_\text{a}$$

$$\varepsilon_{\text{tenv}}\ [[\underline{\text{cond}}(\text{te}_1,\ \text{te}_2,\ \text{te}_3)]] = \underline{\lambda}\text{x}_\text{a}[\text{tt}].\ \varepsilon_{\text{tenv}}\ [[\text{te}_1]]\underline{\cdot}\ \text{x}_\text{a} \underline{\Rightarrow} \varepsilon_{\text{tenv}}\ [[\text{te}_2]]\underline{\cdot}\ \text{x}_\text{a},\ \varepsilon_{\text{tenv}}\ [[\text{te}_3]]\underline{\cdot}\ \text{x}_\text{a}$$
$$\quad \text{where tenv} \vdash \text{te}_1\text{: tt} \underline{\rightarrow} \underline{\text{A}}_{\text{bool}}$$

$$\varepsilon_{\text{tenv}}\ [[\underline{\text{const}}[\text{tt}]\ \text{te}]] = \underline{\lambda}\text{x}_\text{a}[\text{tt}].\ \varepsilon_{\text{tenv}}\ [[\text{te}]]$$

$$\varepsilon_{\text{tenv}}\ [[\underline{\text{id}}[\text{tt}]]] = \underline{\lambda}\text{x}_\text{a}[\text{tt}].\ \text{x}_\text{a}$$

In these rules we have taken the liberty of assuming that x_a and x_b are not in the domain of tenv. To be precise we should have replaced a by m+1 and b by m+2 where m is the largest index i such that x_i is in the domain of tenv. We may note that

Fact 9 The transformation ε preserves the types, that is, if tenv $\vdash$ te:tt holds in TML_m then tenv $\vdash$ $\varepsilon_{\text{tenv}}$ [[te]]: tt will hold in TML_e. □

Hopefully it is intuitively clear that the semantics is preserved.

4 2-level λ-lifting

The translation from TML$_e$ to TML$_m$, i.e. 2-level λ-lifting, is not so straight-forward. To illustrate the problem consider the well-formed TML$_e$ expression

$$\underline{\lambda} x_1[A_1 \underline{\rightarrow} A_1].(\overline{\lambda} x_2[A_1 \underline{\rightarrow} A_1].x_1)\,\overline{\cdot}\, x_1 \tag{1}$$

Although this intuitively will be equivalent to $\underline{\lambda} x_1[A_1 \underline{\rightarrow} A_1].x_1$ that corresponds to $\underline{\mathrm{id}}[A_1 \underline{\rightarrow} A_1]$ in TML$_m$ it would seem that one cannot translate (1) in a compositional way into a combinator expression of TML$_m$. The problem is that a variable, here x_1, that is bound by a (underlined) $\underline{\lambda}$ is passed inside the scope of overlined operators, here $\overline{\lambda}$ and $\overline{\cdot}$. It would seem that there are no ingredients in TML$_m$ that could facilitate this. In fact, by looking at the definition of ε in the previous section it would appear that an expression like (1) will never be produced.

To make these rather vague impressions more precise we shall define a suitable subset TML$_l$ of TML$_e$. This subset will be defined so as not to allow that $\underline{\lambda}$-bound variables are used inside the scope of overlined operators (although there will be one exception). It will emerge that the expression (1) will not be a well-formed expression of TML$_l$. We shall show that ε of the previous section only produces well-formed expressions of TML$_l$ and that λ-lifting is possible when we restrict our attention to TML$_l$. This then will be the formal version of our claim that for 2-level λ-notations the λ-calculus (TML$_e$) and the combinator language (TML$_m$) are not equally expressive.

The types and expressions of TML$_l$ are as for TML$_e$ and we only define a new well-formedness predicate. Since we must distinguish between the variables bound by $\underline{\lambda}$ and those bound by $\overline{\lambda}$ the well-formedness predicate will have the form

$$\text{cenv, renv} \vdash \text{te:tt}$$

where cenv is the type environment for $\overline{\lambda}$-bound variables and renv is the one for $\underline{\lambda}$-bound variables. We shall enforce that the domains of cenv and renv are disjoint. The rules are adapted from those for TML$_e$ by "emptying" the type environment for $\underline{\lambda}$-bound variables when we pass inside the "scope" of (most) overlined operators. They are

$$\text{cenv, renv} \vdash \text{f}_i[\text{tt}]\text{:tt} \qquad \text{if } \exists \text{k}.\ \vdash \text{tt:k and dom(cenv)} \cap \text{dom(renv)} = \emptyset$$

$$\text{cenv, renv} \vdash \text{x}_i\text{: tt} \qquad \text{if } \exists \text{k}.\ \vdash \text{tt:k, cenv(x}_i\text{)=tt or renv(x}_i\text{)=tt and dom(cenv)} \cap \text{dom(renv)} = \emptyset$$

$$\frac{\text{cenv}, \emptyset \vdash \text{te}_1\text{:tt}_1,\ \text{cenv}, \emptyset \vdash \text{te}_2\text{:tt}_2}{\text{cenv, renv} \vdash \overline{(}\text{te}_1, \text{te}_2\overline{)}\text{:tt}_1 \overline{\times} \text{tt}_2} \qquad \text{if} \vdash \text{tt}_1\text{:c and} \vdash \text{tt}_2\text{:c}$$

$$\frac{\text{cenv}, \emptyset \vdash \text{te:tt}_1 \overline{\times} \text{tt}_2}{\text{cenv, renv} \vdash \text{te} \overline{\downarrow} \text{j:tt}_j} \qquad \text{if j} = 1,2$$

$$\frac{\text{cenv}[\text{x}_i \mapsto \text{tt}], \emptyset \vdash \text{te:tt'}}{\text{cenv, renv} \vdash \overline{\lambda}\text{x}_i[\text{tt}].\text{te:tt} \overline{\rightarrow} \text{tt'}} \qquad \text{if} \vdash \text{tt:c and} \vdash \text{tt':c}$$

$$\frac{\text{cenv}, \emptyset \vdash \text{te}_1\text{:tt'} \overline{\rightarrow} \text{tt},\ \text{cenv}, \emptyset \vdash \text{te}_2\text{:tt'}}{\text{cenv, renv} \vdash \text{te}_1\ \overline{\cdot}\ \text{te}_2\text{:tt}}$$

$$\frac{\text{cenv}, \emptyset \vdash \text{te:tt} \overline{\rightarrow} \text{tt}}{\text{cenv, renv} \vdash \overline{\text{fix}}\ \text{te:tt}}$$

$$\frac{\text{cenv}, \emptyset \vdash \text{te:}\overline{\text{A}}_{\text{bool}},\ \text{cenv, renv} \vdash \text{te}_1\text{:tt},\ \text{cenv, renv} \vdash \text{te}_2\text{:tt}}{\text{cenv, renv} \vdash \text{te} \overline{\rightarrow} \text{te}_1, \text{te}_2\text{:tt}}$$

$$\frac{\text{cenv, renv} \vdash \text{te}_1\text{:tt}_1,\ \text{cenv, renv} \vdash \text{te}_2\text{:tt}_2}{\text{cenv, renv} \vdash \underline{(}\text{te}_1, \text{te}_2\underline{)}\text{:tt}_1 \underline{\times} \text{tt}_2} \qquad \text{if} \vdash \text{tt}_1\text{:r and} \vdash \text{tt}_2\text{:r}$$

$$\frac{\text{cenv, renv} \vdash \text{te:tt}_1 \underline{\times} \text{tt}_2}{\text{cenv, renv} \vdash \text{te} \underline{\downarrow j}\text{:tt}_j} \qquad \text{if j = 1,2}$$

$$\frac{\text{cenv}[(\text{dom(cenv)}-\{x_i\}), \text{renv}[x_i \mapsto \text{tt}] \vdash \text{te:tt'}}{\text{cenv, renv} \vdash \underline{\lambda} x_i[\text{tt}].\text{te:tt}\underline{\rightarrow}\text{tt'}} \qquad \text{if} \vdash \text{tt:r and} \vdash \text{tt':r}$$

$$\frac{\text{cenv, renv} \vdash \text{te}_1\text{:tt'}\underline{\rightarrow}\text{tt, cenv, renv} \vdash \text{te}_2\text{:tt'}}{\text{cenv, renv} \vdash \text{te}_1 \;\underline{\cdot}\; \text{te}_2\text{:tt}}$$

$$\frac{\text{cenv, renv} \vdash \text{te:tt}\underline{\rightarrow}\text{tt}}{\text{cenv, renv} \vdash \underline{\text{fix}}\ \text{te:tt}}$$

$$\frac{\text{cenv, renv} \vdash \text{te:}\underline{\text{A}}_{\text{bool}}\text{, cenv, renv} \vdash \text{te}_1\text{:tt, cenv, renv} \vdash \text{te}_2\text{:tt}}{\text{cenv, renv} \vdash \text{te} \underline{\rightarrow} \text{te}_1, \text{te}_2\text{:tt}} \qquad \text{if} \vdash \text{tt:r}$$

In the rule for overlined conditional we have not emptied the type environment for $\underline{\lambda}$-bound variables for the "then" and "else" branches. This is connected with the fact that we have not restricted the type of the conditional to be of compile-time type and this is of importance when we come to λ-lifting. In the rule for underlined conditional we have restricted the type of the conditional to be of run-time type. This is in accord with the restrictions implicit in the rule for $\underline{\text{cond}}$ in TML$_m$. Finally we note that in analogy with Fact 7 and 8 we have

Fact 10 Expressions have well-formed types, i.e. if cenv, renv $\vdash$ te:tt then $\exists$k. $\vdash$ tt:k. □

Fact 11 Expressions are uniquely typed, i.e. if cenv, renv $\vdash$ te:tt$_1$ and cenv, renv $\vdash$ te:tt$_2$ then tt$_1$=tt$_2$. □

In Fact 10 (and similarly in Fact 7) the kind k will be unique unless tt is of the form tt$_1$ $\underline{\rightarrow}$ tt$_2$ in which case k may be c as well as r. It is straightforward to verify that the expression (1) is not well-formed in TML$_l$. It is also easy to see that any expression that is well-formed in TML$_l$ also will be well-formed in TML$_e$. That we have not gone too far in the definition of TML$_l$ may be expressed by

Fact 12 The expansion ε of combinators in TML$_m$ only produces expressions in TML$_l$, i.e. if tenv$\vdash$ te:tt in TML$_m$ then tenv, $\emptyset \vdash \varepsilon_{\text{tenv}}$ [[te]]:tt in TML$_l$. □

Turning to 2-level λ-lifting the intention will be to define a function

$$\Lambda^{\text{cenv}}_{\text{penv}}\text{: \{ te} \in \text{TML}_l \mid \exists\text{tt. cenv}, \rho(\text{penv}) \vdash \text{te:tt \}} \rightarrow \text{\{ te} \mid \text{te} \in \text{TML}_m \text{ \}}$$

whenever cenv is a type environment, penv is a position environment and the domains of cenv and ρ(penv) are disjoint. Looking back to the 1-level case we note that there we demanded that the position environment could not be the empty list. It is evident from the well-formedness rules for TML$_l$ that the emptying of the type environment for $\underline{\lambda}$-bound variables means that we cannot take a similar cavalier attitude here. So the idea will be to demand that cenv, ρ(penv) $\vdash$ te:tt satisfies that

$$\text{penv} = () \Rightarrow\ \vdash \text{tt:c}$$

$$\text{penv} \neq () \Rightarrow\ \vdash \text{tt:r}$$

This means that when tt is tt$_1$ $\underline{\rightarrow}$tt$_2$ the position environment will tell us whether we want to regard tt$_1$ $\underline{\rightarrow}$ tt$_2$ as a run-time data object (in case penv $\neq$ ()) or as a run-time computation (in case penv=()). Thus we do not need to follow D.Schmidt [12] in making these distinctions in a syntactic manner (by having essentially three kinds of function arrows rather than just two).

The definition of $\Lambda^{\text{cenv}}_{\text{penv}}$[[te]] will closely follow the inference that cenv,ρ(penv)$\vdash$ te:tt for some tt. We have already pointed out that in this inference we shall replace the type environment ρ(penv) by the empty map $\emptyset$ when we move inside the "scope" of (most) overlined operators. In the definition of

$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}[\![\mathrm{te}]\!]$ the analogue will be to replace the position environment penv by the empty list (). Since this will entail ignoring some (run-time) arguments we define

$$\delta(\mathrm{penv})\ \mathrm{te} = \begin{cases} \mathrm{te} & \text{if penv} = () \\ \underline{\mathrm{const}}[\Pi(\mathrm{penv})]\ \mathrm{te} & \text{if penv} \neq () \end{cases}$$

where $\Pi(\mathrm{penv})$ is defined as in Section 2 but now uses $\underline{\times}$ in stead of $\times$. If we also define

$$\Delta(\mathrm{penv},\mathrm{tt}) = \begin{cases} \mathrm{tt} & \text{if penv} = () \\ \Pi(\mathrm{penv}) \underline{\rightarrow} \mathrm{tt} & \text{if penv} \neq () \end{cases}$$

we may note that if tenv $\vdash$ te:tt holds in TML_m then also tenv $\vdash$ $\delta(\mathrm{penv})$ te:$\Delta(\mathrm{penv},\mathrm{tt})$ holds in TML_m.

Similarly we may move into the "scope" of an underlined operator and there we cannot allow the position environment to be the empty list (as is witnessed by our insistence on penv $\neq$ () in the case of 1-level λ-lifting). This motivates defining

$$\varphi(\mathrm{tt},\mathrm{penv}) = \begin{cases} \mathrm{penv} & \text{if penv} \neq () \\ ((\mathrm{x_a},\mathrm{tt})) & \text{if penv} = () \end{cases}$$

for a dummy variable $\mathrm{x_a}$ so that $\varphi(\mathrm{tt},\mathrm{penv})$ is never the empty list. In connection with this we need

$$\omega(\mathrm{tt},\mathrm{penv})\ \mathrm{te} = \begin{cases} \mathrm{te} & \text{if penv} \neq () \\ \underline{\mathrm{apply}}[\mathrm{tt}_1 \underline{\rightarrow} \mathrm{tt}_2] \ \Box\ \underline{\mathrm{tuple}}(\mathrm{te},\underline{\mathrm{id}}[\mathrm{tt}_1]) & \\ & \text{if penv} = () \text{ and tt} = \mathrm{tt}_1 \underline{\rightarrow} \mathrm{tt}_2 \end{cases}$$

where we shall take care that the first argument to ω always will have the proper form. One may note that if penv = () and $\emptyset \vdash \mathrm{te}{:}\mathrm{tt}_1 \underline{\rightarrow} \mathrm{tt}_1 \underline{\rightarrow} \mathrm{tt}_2$ then $\emptyset \vdash \omega(\mathrm{tt}_1 \underline{\rightarrow} \mathrm{tt}_2,\mathrm{penv})\ \mathrm{te}{:}\mathrm{tt}_1 \underline{\rightarrow} \mathrm{tt}_2$ and it will be clear from below that ω will be used when we want to escape from the effects of having used φ. (This will be done by taking an argument and supplying it to the function twice so that the additional argument needed because of the introduction of the dummy variable will be catered for.)

Turning to the definition of $\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}$ we use a version of $\pi^{\mathrm{penv}}_{\mathrm{j}}$ that is as in Section 2 but that uses underlined combinators. It may be helpful to take a look at Fact 13 while reading through the following equations

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{f_i}[\mathrm{tt}]]\!] = \delta(\mathrm{penv})\ \mathrm{f_i}[\mathrm{tt}]$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{x_i}]\!] = \begin{cases} \delta(\mathrm{penv})\ \mathrm{x_i} & \text{if } \mathrm{x_i} \in \mathrm{dom(cenv)} \\ \pi^{\mathrm{penv}}_{\mathrm{i}} & \text{if } \mathrm{x_i} \in \mathrm{dom(penv)} \end{cases}$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\overline{(}\mathrm{te}_1,\ \mathrm{te}_2\overline{)}]\!] = \overline{(}\Lambda^{\mathrm{cenv}}_{()}\ [\![\mathrm{te}_1]\!],\ \Lambda^{\mathrm{cenv}}_{()}\ [\![\mathrm{te}_2]\!]\overline{)}$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te}\ \overline{\downarrow \mathrm{j}}]\!] = \delta(\mathrm{penv})\ (\Lambda^{\mathrm{cenv}}_{()}\ [\![\mathrm{te}]\!]\overline{\downarrow \mathrm{j}})$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\overline{\lambda}\mathrm{x_i}[\mathrm{tt}].\mathrm{te}]\!] = \overline{\lambda}\mathrm{x_i}[\mathrm{tt}].\Lambda^{\mathrm{cenv}[\mathrm{x_i}\mapsto\mathrm{tt}]}_{()}\ [\![\mathrm{te}]\!]$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te}_1\ \overline{\cdot}\ \mathrm{te}_2]\!] = \delta(\mathrm{penv})\ (\Lambda^{\mathrm{cenv}}_{()}\ [\![\mathrm{te}_1]\!]\overline{\cdot}\ \Lambda^{\mathrm{cenv}}_{()}\ [\![\mathrm{te}_2]\!])$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\overline{\mathrm{fix}}\ \mathrm{te}]\!] = \delta(\mathrm{penv})\ (\overline{\mathrm{fix}}\ \Lambda^{\mathrm{penv}}_{()}\ [\![\mathrm{te}]\!])$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te}_1\ \overline{\Rightarrow}\ \mathrm{te}_2,\ \mathrm{te}_3]\!] = \Lambda^{\mathrm{cenv}}_{()}\ [\![\mathrm{te}_1]\!]\overline{\Rightarrow}\ \Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te}_2]\!],\ \Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te}_3]\!]$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\underline{(}\mathrm{te}_1,\ \mathrm{te}_2\underline{)}]\!] = \underline{\mathrm{tuple}}(\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te}_1]\!],\ \Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te}_2]\!])$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te}\ \underline{\downarrow \mathrm{j}}]\!] = \omega(\mathrm{tt_j},\mathrm{penv})(\underline{\mathrm{take}}_{\mathrm{j}}[\mathrm{tt}_1 \underline{\times} \mathrm{tt}_2]\ \Box\ \Lambda^{\mathrm{cenv}}_{\varphi(\mathrm{tt_j},\mathrm{penv})}\ [\![\mathrm{te}]\!])$$

where cenv, $\rho(\mathrm{penv}) \vdash \mathrm{te}{:}\mathrm{tt}_1 \underline{\times} \mathrm{tt}_2$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\underline{\lambda}\mathrm{x_i}[\mathrm{tt}].\mathrm{te}]\!]= \begin{cases} \Lambda^{\mathrm{cenv}\lceil(\mathrm{dom(cenv)}-\{\mathrm{x_i}\})}_{((\mathrm{x_i},\mathrm{tt}))}\ [\![\mathrm{te}]\!] & \text{if penv} = () \\ \underline{\mathrm{curry}}\ (\Lambda^{\mathrm{cenv}\lceil(\mathrm{dom(cenv)}-\{\mathrm{x_i}\})}_{((\mathrm{x_i},\mathrm{tt}))\hat{}\,\mathrm{penv}}\ [\![\mathrm{te}]\!]\ ) & \text{if penv} \neq () \end{cases}$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te_1}\ \underline{:}\ \mathrm{te_2}]\!]= \omega(\mathrm{tt_2},\mathrm{penv})(\underline{\mathrm{apply}}[\mathrm{tt_1}\underline{\rightarrow}\mathrm{tt_2}]\ \Box\ \underline{\mathrm{tuple}}(\Lambda^{\mathrm{cenv}}_{\varphi(\mathrm{tt_2},\mathrm{penv})}\ [\![\mathrm{te_1}]\!],\ \Lambda^{\mathrm{cenv}}_{\varphi(\mathrm{tt_2},\mathrm{penv})}\ [\![\mathrm{te_2}]\!]))$$

where cenv, ρ(penv) $\vdash$ $\mathrm{te_1}:\mathrm{tt_1}\underline{\rightarrow}\mathrm{tt_2}$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\underline{\mathrm{fix}}\ \mathrm{te}]\!]= \omega(\mathrm{tt},\mathrm{penv})(\underline{\mathrm{fix}}[\mathrm{tt}]\ \Box\ \Lambda^{\mathrm{cenv}}_{\varphi(\mathrm{tt},\mathrm{penv})}\ [\![\mathrm{te}]\!]) \qquad \text{where cenv, } \rho(\mathrm{penv}) \vdash \mathrm{te}:\mathrm{tt}\underline{\rightarrow}\mathrm{tt}$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te_1} \underline{\rightarrow}\ \mathrm{te_2},\ \mathrm{te_3}]\!]= \omega(\mathrm{tt},\mathrm{penv})\ \underline{\mathrm{cond}}(\Lambda^{\mathrm{cenv}}_{\varphi(\mathrm{tt},\mathrm{penv})}\ [\![\mathrm{te_1}]\!],\ \Lambda^{\mathrm{cenv}}_{\varphi(\mathrm{tt},\mathrm{penv})}\ [\![\mathrm{te_2}]\!],\ \Lambda^{\mathrm{cenv}}_{\varphi(\mathrm{tt},\mathrm{penv})}\ [\![\mathrm{te_3}]\!])$$

where cenv, ρ(penv) $\vdash$ $\mathrm{te_2}:\mathrm{tt}$

In these equations the intention is that the greek-letter operations should be "macro-expanded" as they are really a shorthand for their defining equation and are not parts of TML_m. To illustrate the use of δ we shall consider the expression

$$\underline{\lambda}\mathrm{x_1}[\underline{\mathrm{A_1}\rightarrow\mathrm{A_1}}].\mathrm{f_1}[\underline{\mathrm{A_1}\rightarrow\mathrm{A_1}}]$$

When processing $\mathrm{f_1}[\underline{\mathrm{A_1}\rightarrow\mathrm{A_1}}]$ the position environment will not be empty and so we must use the construct $\underline{\mathrm{const}}[\underline{\mathrm{A_1}\rightarrow\mathrm{A_1}}]$ (via δ) to get rid of the run-time argument corresponding to $\mathrm{x_1}$. To illustrate the use of ω we shall consider the expression

$$\underline{(}\underline{\lambda}\mathrm{x_1}[\underline{\mathrm{A_1}}].\mathrm{x_1},\ \underline{\lambda}\mathrm{x_1}[\underline{\mathrm{A_1}}].\mathrm{x_1}\ \underline{)\!\downarrow 1}$$

which is well-formed and of the compile-time type $\underline{\mathrm{A_1}\rightarrow\mathrm{A_1}}$. So it would be natural to use an empty position environment but then we cannot sequence the operations. (This is the same problem as in Section 2.) Consequently, we use ω and φ to translate the expression as if the position environment had not been empty and then later to get rid of the extra element in the position environment. It is vital for this technique to succeed that if $\vdash$ tt:k holds for k = c as well as k = r then tt is of the form $\mathrm{tt_1}\underline{\rightarrow}\mathrm{tt_2}$ as may be seen from the definition of ω. By way of digression it is worth observing that $\delta(\cdots)$ te roughly corresponds to $\mathrm{te}^{>}$ in [1] and similarly $\omega(\cdots)$ te roughly corresponds to $\mathrm{te}^{<}$.

The relationship between an expression and its λ-lifted version is given by

Fact 13 Whenever the domains of cenv and ρ(penv) are disjoint the above equations define a function of the stated functionality and it satisfies

$$\mathrm{cenv} \vdash \Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![\mathrm{te}]\!]:\Delta(\mathrm{penv},\mathrm{tt})$$

whenever cenv,ρ(penv) $\vdash$ te:tt and penv = () $\Rightarrow$ $\vdash$ tt:c and penv $\neq$ () $\Rightarrow$ $\vdash$ tt:r. $\Box$

Furthermore we shall claim that the semantics has been preserved. The importance of this fact is that the combinator notation of TML_m is equivalent in expressive power to the proper subset TML_l of the λ-notation TML_e.

Example: As an example of the use of the 2-level λ-lifting consider the TML_l expression

$$\begin{aligned} \mathrm{select'} \equiv\ & \overline{\mathrm{fix}}(\overline{\lambda}\mathrm{S}[\overline{\mathrm{Int}}\overline{\rightarrow}\underline{\mathrm{List}\rightarrow\mathrm{Int}}].\ \overline{\lambda}\mathrm{n}[\overline{\mathrm{Int}}].\ \underline{\lambda}\mathrm{l}[\underline{\mathrm{List}}]. \\ & \qquad (=[\overline{\mathrm{Int}}\overline{\rightarrow}\overline{\mathrm{Int}}\overline{\rightarrow}\overline{\mathrm{Int}}]\overline{\cdot}\ 1[\overline{\mathrm{Int}}]\overline{\cdot}\ \mathrm{n})\overline{\rightarrow}\ (\mathrm{hd}[\underline{\mathrm{List}\rightarrow\mathrm{Int}}]\underline{:}\ \mathrm{l}), \\ & \qquad\quad (\mathrm{S}\overline{\cdot}\ (-[\overline{\mathrm{Int}}\overline{\rightarrow}\overline{\mathrm{Int}}\overline{\rightarrow}\overline{\mathrm{Int}}]\overline{\cdot}\ \mathrm{n}\overline{\cdot}\ 1[\overline{\mathrm{Int}}])\underline{:}\ (\mathrm{tl}[\underline{\mathrm{List}\rightarrow\mathrm{List}}]\underline{:}\ \mathrm{l}))) \end{aligned}$$

where we e.g. write $\overline{\mathrm{Int}}$ for $\overline{\mathrm{A}}_{\mathrm{Int}}$, $1[\overline{\mathrm{Int}}]$ for $\mathrm{f_1}[\overline{\mathrm{A}}_{\mathrm{Int}}]$ and n for $\mathrm{x_n}$. First we get

$$\Lambda^{\emptyset}_{()}[\![\mathrm{select'}]\!] = \overline{\mathrm{fix}}\ \Lambda^{\emptyset}_{()}[\![\overline{\lambda}\mathrm{S}[\ldots].\ldots]\!]$$

because the initial position environment is empty. The equation for compile-time λ-abstraction then gives

$$\begin{aligned}&\Lambda^{\emptyset}_{()}[\![\overline{\lambda}\mathrm{S}[\overline{\mathrm{Int}}\overline{\Rightarrow}\underline{\mathrm{List}\rightarrow\mathrm{Int}}].\ \overline{\lambda}\mathrm{n}[\overline{\mathrm{Int}}].\ \underline{\lambda}\mathrm{l}[\underline{\mathrm{List}}].\ldots]\!] =\\ &\quad \overline{\lambda}\mathrm{S}[\overline{\mathrm{Int}}\overline{\Rightarrow}\underline{\mathrm{List}\rightarrow\mathrm{Int}}].\ \Lambda^{[\mathrm{S}\mapsto\ \ldots]}_{()}\ [\![\overline{\lambda}\mathrm{n}[\overline{\mathrm{Int}}].\ \underline{\lambda}\mathrm{l}[\underline{\mathrm{List}}].\ldots]\!] =\\ &\quad \overline{\lambda}\mathrm{S}[\overline{\mathrm{Int}}\overline{\Rightarrow}\underline{\mathrm{List}\rightarrow\mathrm{Int}}].\ \overline{\lambda}\mathrm{n}[\overline{\mathrm{Int}}].\ \Lambda^{\mathrm{cenv}}_{()}\ [\![\underline{\lambda}\mathrm{l}[\underline{\mathrm{List}}].\ldots]\!]\end{aligned}$$

where cenv abbreviates $[\mathrm{S}\mapsto \overline{\mathrm{Int}}\overline{\Rightarrow}\underline{\mathrm{List}\rightarrow\mathrm{Int}};\ \mathrm{n}\mapsto \overline{\mathrm{Int}}]$. Since the position environment still is empty the equation for run-time λ-abstraction gives

$$\Lambda^{\mathrm{cenv}}_{()}\ [\![\underline{\lambda}\mathrm{l}[\underline{\mathrm{List}}].(\ldots)\overline{\Rightarrow}(\ldots),(\ldots)]\!] = \Lambda^{\mathrm{cenv}}_{\mathrm{penv}}\ [\![(\ldots)\overline{\Rightarrow}(\ldots),(\ldots)]\!]$$

where penv is $((\mathrm{l},\underline{\mathrm{List}}))$. Using the equation for compile-time conditional we see that

$$\Lambda^{\mathrm{cenv}}_{()}[\![=[\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}]\bar{\cdot}\ 1[\overline{\mathrm{Int}}]\bar{\cdot}\ \mathrm{n}]\!],$$

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}[\![\mathrm{hd}[\underline{\mathrm{List}\rightarrow\mathrm{Int}}]\underline{\cdot}\ \mathrm{l}]\!]$$

and

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}[\![\mathrm{S}\bar{\cdot}\ (-[\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}]\bar{\cdot}\ \mathrm{n}\bar{\cdot}\ 1[\overline{\mathrm{Int}}])\underline{\cdot}\ (\mathrm{tl}[\underline{\mathrm{List}\rightarrow\mathrm{List}}]\underline{\cdot}\ \mathrm{l})]\!]$$

must be calculated.

For the condition we get

$$\begin{aligned}&\Lambda^{\mathrm{cenv}}_{()}[\![=[\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}]\bar{\cdot}\ 1[\overline{\mathrm{Int}}]\bar{\cdot}\ \mathrm{n}]\!] = \Lambda^{\mathrm{cenv}}_{()}[\![=[\ldots]]\!]\bar{\cdot}\ \Lambda^{\mathrm{cenv}}_{()}[\![1[\overline{\mathrm{Int}}]]\!]\bar{\cdot}\ \Lambda^{\mathrm{cenv}}_{()}[\![\mathrm{n}]\!] =\\ &\quad =[\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}]\bar{\cdot}\ 1[\overline{\mathrm{Int}}]\bar{\cdot}\ \mathrm{n}\end{aligned}$$

so the expression is unchanged. For the true-branch of the conditional we get

$$\begin{aligned}&\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}[\![\mathrm{hd}[\underline{\mathrm{List}\rightarrow\mathrm{Int}}]\underline{\cdot}\ \mathrm{l}]\!] = \underline{\mathrm{apply}}[\underline{\mathrm{List}\rightarrow\mathrm{Int}}]\ \square\ \underline{\mathrm{tuple}}(\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}[\![\mathrm{hd}[\ldots]]\!],\ \Lambda^{\mathrm{cenv}}_{\mathrm{penv}}[\![\mathrm{l}]\!]) =\\ &\quad \underline{\mathrm{apply}}[\underline{\mathrm{List}\rightarrow\mathrm{Int}}]\ \square\ \underline{\mathrm{tuple}}(\underline{\mathrm{const}}[\underline{\mathrm{List}}]\ \mathrm{hd}[\underline{\mathrm{List}\rightarrow\mathrm{Int}}],\underline{\mathrm{id}}[\underline{\mathrm{List}}])\end{aligned}$$

The false-branch of the conditional is more interesting. Similar to above we get

$$\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}[\![\mathrm{tl}[\underline{\mathrm{List}\rightarrow\mathrm{List}}]\underline{\cdot}\ \mathrm{l}]\!] = \underline{\mathrm{apply}}[\underline{\mathrm{List}\rightarrow\mathrm{List}}]\ \square\ \underline{\mathrm{tuple}}(\underline{\mathrm{const}}[\underline{\mathrm{List}}]\ \mathrm{tl}[\underline{\mathrm{List}\rightarrow\mathrm{List}}],\underline{\mathrm{id}}[\underline{\mathrm{List}}])$$

Concerning $\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}[\![\mathrm{S}\bar{\cdot}\ (-[\ldots]\bar{\cdot}\ \mathrm{n}\bar{\cdot}\ 1[\ldots])]\!]$ the expression should be computed at compile-time but the position environment is non-empty so it is supposed to take a run-time parameter. We use the $\underline{\mathrm{const}}$ expression (via δ) to get rid of the run-time parameter:

$$\begin{aligned}&\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}[\![\mathrm{S}\bar{\cdot}\ (-[\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}]\bar{\cdot}\ \mathrm{n}\bar{\cdot}\ 1[\overline{\mathrm{Int}}])]\!] =\\ &\quad \underline{\mathrm{const}}[\underline{\mathrm{List}}]\ (\Lambda^{\mathrm{cenv}}_{()}[\![\mathrm{S}]\!]\bar{\cdot}\ \Lambda^{\mathrm{cenv}}_{()}[\![-[\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}]\bar{\cdot}\ \mathrm{n}\bar{\cdot}\ 1[\overline{\mathrm{Int}}]]\!])\end{aligned}$$

and it is now straightforward to show that

$$\Lambda^{\mathrm{cenv}}_{()}[\![\mathrm{S}]\!] = \mathrm{S}$$

and

$$\Lambda^{\mathrm{cenv}}_{()}[\![-[\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}]\bar{\cdot}\ \mathrm{n}\bar{\cdot}\ 1[\overline{\mathrm{Int}}]]\!] = -[\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}]\bar{\cdot}\ \mathrm{n}\bar{\cdot}\ 1[\overline{\mathrm{Int}}]$$

so for the false-branch we get

$$\begin{aligned}&\Lambda^{\mathrm{cenv}}_{\mathrm{penv}}[\![\mathrm{S}\bar{\cdot}\ (-[\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}]\bar{\cdot}\ \mathrm{n}\bar{\cdot}\ 1[\overline{\mathrm{Int}}])\underline{\cdot}\ (\mathrm{tl}[\underline{\mathrm{List}\rightarrow\mathrm{List}}]\underline{\cdot}\ \mathrm{l})]\!] =\\ &\quad \underline{\mathrm{apply}}[\underline{\mathrm{List}\rightarrow\mathrm{Int}}]\ \square\ \underline{\mathrm{tuple}}(\underline{\mathrm{const}}[\underline{\mathrm{List}}]\ (\mathrm{S}\bar{\cdot}\ (-[\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}\overline{\Rightarrow}\overline{\mathrm{Int}}]\bar{\cdot}\ \mathrm{n}\bar{\cdot}\ 1[\overline{\mathrm{Int}}])),\\ &\qquad\qquad \underline{\mathrm{apply}}[\underline{\mathrm{List}\rightarrow\mathrm{List}}]\ \square\ \underline{\mathrm{tuple}}(\underline{\mathrm{const}}[\underline{\mathrm{List}}]\ \mathrm{tl}[\underline{\mathrm{List}\rightarrow\mathrm{List}}],\underline{\mathrm{id}}[\underline{\mathrm{List}}]))\end{aligned}$$

Putting things together we get

$$\begin{array}{l}
\Lambda^{\emptyset}_{()}[\![\text{select'}]\!] = \\
\quad \overline{\text{fix}}(\overline{\lambda}\text{S}[\overline{\text{Int}\Rightarrow}\underline{\text{List}\rightarrow\text{Int}}].\ \overline{\lambda}\text{n}[\overline{\text{Int}}].\ (\overline{=}[\overline{\text{Int}}\overline{\Rightarrow}\overline{\text{Int}}\overline{\Rightarrow}\overline{\text{Int}}]\,\overline{\cdot}\ 1[\overline{\text{Int}}]\,\overline{\cdot}\ \text{n})\overline{\Rightarrow} \\
\qquad\quad \underline{\text{apply}}[\underline{\text{List}\rightarrow\text{Int}}] \ \square\ \underline{\text{tuple}}(\underline{\text{const}}[\underline{\text{List}}]\ \text{hd}[\underline{\text{List}\rightarrow\text{Int}}],\underline{\text{id}}[\underline{\text{List}}]), \\
\qquad\quad \underline{\text{apply}}[\underline{\text{List}\rightarrow\text{Int}}] \ \square\ \underline{\text{tuple}}(\underline{\text{const}}[\underline{\text{List}}]\ (\text{S}\,\overline{\cdot}\ (\overline{-}[\overline{\text{Int}}\overline{\Rightarrow}\overline{\text{Int}}\overline{\Rightarrow}\overline{\text{Int}}]\,\overline{\cdot}\ \text{n}\,\overline{\cdot}\ 1[\overline{\text{Int}}])), \\
\qquad\qquad\quad \underline{\text{apply}}[\underline{\text{List}\rightarrow\text{List}}] \ \square\ \underline{\text{tuple}}(\underline{\text{const}}[\underline{\text{List}}]\ \text{tl}[\underline{\text{List}\rightarrow\text{List}}],\underline{\text{id}}[\underline{\text{List}}])))
\end{array}$$

The λ-lifting is specified in a syntax directed manner and this is the reason for the complicated form of the expression above. Semantically

$$\underline{\text{apply}}[\text{tt}\underline{\rightarrow}\text{tt'}] \ \square\ \underline{\text{tuple}}(\underline{\text{const}}[\text{tt}]\ \text{te},\underline{\text{id}}[\text{tt}]) \equiv \text{te}$$

and

$$\underline{\text{apply}}[\text{tt}\underline{\rightarrow}\text{tt'}] \ \square\ \underline{\text{tuple}}(\underline{\text{const}}[\text{tt}]\ \text{te},\text{te'}) \equiv \text{te} \ \square\ \text{te'}$$

so by performing a little partial evaluation [10] we can replace the expression above by

$$\begin{array}{l}
\overline{\text{fix}}(\overline{\lambda}\text{S}[\overline{\text{Int}\Rightarrow}\underline{\text{List}\rightarrow\text{Int}}].\ \overline{\lambda}\text{n}[\overline{\text{Int}}].\ (\overline{=}[\overline{\text{Int}}\overline{\Rightarrow}\overline{\text{Int}}\overline{\Rightarrow}\overline{\text{Int}}]\,\overline{\cdot}\ 1[\overline{\text{Int}}]\,\overline{\cdot}\ \text{n})\overline{\Rightarrow} \\
\qquad\qquad \text{hd}[\underline{\text{List}\rightarrow\text{Int}}],\ (\text{S}\,\overline{\cdot}\ (\overline{-}[\overline{\text{Int}}\overline{\Rightarrow}\overline{\text{Int}}\overline{\Rightarrow}\overline{\text{Int}}]\,\overline{\cdot}\ \text{n}\,\overline{\cdot}\ 1[\overline{\text{Int}}]))\ \square\ \text{tl}[\underline{\text{List}\rightarrow\text{List}}])
\end{array}$$

which is the expression called select" in the Introduction.

5 Applications in denotational semantics

In our previous work [7] we have studied the application of a variant of TML_m as a meta-language for denotational semantics. One motivation for this is that the informal semantics of a language often makes a distinction between those computations that are performed at compile-time and those that are performed at run-time. An example is Tennent's distinction between static expression procedures and expression procedures in Pascal-like languages [14]. Another motivation is that the automatic generation of compilers from denotational definitions will benefit from the distinction between binding times in that code only will be generated for the run-time level [8].

In [7] we study a Pascal-like language with e.g. declarations, commands and expressions. We do not have the space to give all the details so let us concentrate on the important semantic domains:

Dv	$= \text{A}_{\text{Loc}} + \cdots$	denotable values
Env	$= \text{A}_{\text{Ide}} \rightarrow \text{Dv}$	environments
Ev	$= \text{A}_{\text{Loc}} + \cdots$	expressible values
S	$= \text{A}_{\text{Loc}} \rightarrow \cdots$	stores
Ans	$= \cdots$	answers
Cc	$= \text{S} \rightarrow \text{Ans}$	command continuations
Ec	$= \text{Ev} \rightarrow \text{Cc}$	expression continuations

As an example, in a standard continuation style semantics the semantic function $\mathcal{E}$ for expressions (Exp) could have the functionality

$$\mathcal{E}\text{: Exp} \rightarrow \text{Env} \rightarrow \text{Ec} \rightarrow \text{Cc}$$

As mentioned above we want to distinguish between compile-time and run-time. The idea will therefore be to rewrite the semantics using the 2-level meta-language TML_e because this will make the binding times explicit. The operational intuition is that environments and denotable values will be compile-time objects whereas stores, answers and expressible values will be run-time objects. This gives rise to the following semantic domains

$$\begin{array}{lll}
\mathrm{Dv} & = \overline{\mathrm{A}}_{\mathrm{Loc}} \mathbin{\overline{+}} \overline{\cdots} & \text{(kind c)} \\
\mathrm{Env} & = \overline{\mathrm{A}}_{\mathrm{Ide}} \mathbin{\overline{\rightarrow}} \mathrm{Dv} & \text{(kind c)} \\
\mathrm{Ev} & = \underline{\mathrm{A}}_{\mathrm{Loc}} \mathbin{\underline{+}} \underline{\cdots} & \text{(kind r)} \\
\mathrm{S} & = \underline{\mathrm{A}}_{\mathrm{Loc}} \mathbin{\underline{\rightarrow}} \underline{\cdots} & \text{(kind r)} \\
\mathrm{Ans} & = \underline{\cdots} & \text{(kind r)} \\
\mathrm{Cc} & = \mathrm{S} \mathbin{\underline{\rightarrow}} \mathrm{Ans} & \text{(both kind r and c)} \\
\mathrm{Ec} & = \mathrm{Ev} \mathbin{\underline{\rightarrow}} \mathrm{Cc} & \text{(both kind r and c)}
\end{array}$$

and the semantic function $\mathcal{E}$ will now have the functionality

$$\mathcal{E}\colon \mathrm{Exp} \mathbin{\overline{\rightarrow}} \mathrm{Env} \mathbin{\overline{\rightarrow}} \mathrm{Ec} \mathbin{\overline{\rightarrow}} \mathrm{Cc}$$

The semantic equations of DML$_e$ can now be rewritten in a similar way. As an example consider the following semantic equation (from [7])

$$\mathcal{E}[\![\mathrm{e}_1+\mathrm{e}_2]\!] = \lambda \mathrm{r}[\mathrm{Env}].\lambda \mathrm{k}[\mathrm{Ec}].\mathcal{E}[\![\mathrm{e}_1]\!] \cdot \mathrm{r} \cdot (\lambda \mathrm{v}_1[\mathrm{Ev}].\mathcal{E}[\![\mathrm{e}_2]\!] \cdot \mathrm{r} \cdot (\lambda \mathrm{v}_2[\mathrm{Ev}].\mathrm{k} \cdot (\mathcal{O}[\![+]\!] \cdot (\mathrm{v}_1,\mathrm{v}_2))))$$

where $\mathcal{O}[\![+]\!]$: Ev $\times$ Ev $\rightarrow$ Ev. It may now be replaced by the following semantic equation of TML$_e$:

$$\mathcal{E}[\![\mathrm{e}_1+\mathrm{e}_2]\!] = \overline{\lambda} \mathrm{r}[\mathrm{Env}].\overline{\lambda} \mathrm{k}[\mathrm{Ec}].\mathcal{E}[\![\mathrm{e}_1]\!] \mathbin{\overline{\cdot}} \mathrm{r} \mathbin{\overline{\cdot}} (\underline{\lambda} \mathrm{v}_1[\mathrm{Ev}].\mathcal{E}[\![\mathrm{e}_2]\!] \mathbin{\overline{\cdot}} \mathrm{r} \mathbin{\overline{\cdot}} (\underline{\lambda} \mathrm{v}_2[\mathrm{Ev}].\mathrm{k} \mathbin{\underline{\cdot}} (\mathcal{O}[\![+]\!] \mathbin{\underline{\cdot}} \underline{(}\mathrm{v}_1,\mathrm{v}_2\underline{)})))$$

We note that the environment and the expression continuation are passed as compile-time arguments whereas the "intermediate values" v_1 and v_2 occur at the run-time level. In [11] we have given algorithms that will perform this transformation given information about the 2-level type of the semantic function. We refer to that paper for the details.

In the compiler we shall want to generate code for the run-time computations and the technique of 2-level λ-lifting will show us how to generate combinator code. This code can then be transformed into code for other abstract machines as described in [8]. Thus the remaining step will be to transform the semantic equations in TML$_e$ into semantic equations in TML$_m$. As witnessed by the results of Section 4 this will only be straightforward if the semantic equations are in TML$_l$ rather than TML$_e$. For the example language this will indeed be the case for most semantic equations (see [7]), but the above equation for $\mathcal{E}[\![\mathrm{e}_1+\mathrm{e}_2]\!]$ will be an exception. To see this note that the compile-time function application in the body of $\underline{\lambda}\mathrm{v}_1[\mathrm{Ev}].\cdots \mathbin{\overline{\cdot}} (\underline{\lambda}\mathrm{v}_2[\mathrm{Ev}].\cdots)$ implies that v_1 must not occur free in the subexpression $\underline{\lambda}\mathrm{v}_2[\mathrm{Ev}].\cdots$ but unfortunately it does! So the well-formedness conditions of TML$_l$ are not fulfilled. This then explains that our inability in [7] to provide semantic equations in TML$_m$ is a necessary (although undesired) consequence of the nature of TML$_m$. Thus one would have to study *heuristics* for how to pass from TML$_e$ to TML$_l$. Possible candidates are the rewriting of a continuation style semantics into a direct style semantics (as done in [7]) or the rewriting of a continuation style standard semantics into a continuation style store semantics (in the sense of [5]).

6 Conclusion

We have extended the notion of λ-lifting to 2-level languages using λ-notation at the top-level and λ-notation or combinator-notation at the bottom-level. The development has been performed for a 2-level type system with products and function spaces but recursive types and sum types may be incorporated as well. Unlike the 1-level case it turns out that the natural formulations of the two sorts of 2-level languages are not equally expressive in that the mixed notation (TML$_m$) corresponds to a proper subset of the pure λ-notation (TML$_e$). For the purposes of the implementation of functional languages in general, and semantics directed compiling in particular, this gives a clear formulation of some of the problems involved in achieving efficiency: The efficient implementation of a functional language (or a semantic notation) inevitably involves making a distinction between run-time and compile-time. In [11] it is shown how this distinction may be introduced in a mostly automatic way into a λ-calculus. However, this paper shows that the result (which is an expression in TML$_e$) is not necessarily in a form (namely TML$_l$) where one may vary the interpretations of the combinators (as in e.g. [8]). This calls for further research into how TML$_m$ can be extended so as to correspond more closely to TML$_e$.

References

[1] P.-L.Curien: Categorical Combinators, Sequential Algorithms and Functional Programming, Pitman, London, 1986.

[2] H.B.Curry, R.Feys: Combinatory Logic, vol.1, North-Holland, Amsterdam, 1958.

[3] R.J.M.Hughes: Super-combinators, a new implementation method for applicative languages, Conf. Record of the 1982 ACM Symposium on LISP and functional programming, 1982.

[4] T.Johnson: Lambda lifting – transforming programs to recursive equations, Functional Programming Languages and Computer Architecture, Springer LNCS 201, 1985.

[5] R.Milne, C.Strachey: A Theory of Programming Language Semantics, Halstead Press, 1976.

[6] F.Nielson: Abstract interpretation of denotational definitions, STACS 1986, Springer LNCS 210, 1986.

[7] H.R.Nielson, F.Nielson: Pragmatic aspects of two-level denotational meta-languages, ESOP 1986, Springer LNCS 213, 1986.

[8] H.R.Nielson, F.Nielson: Semantics directed compiling for functional languages, Proceedings of the 1986 ACM Conf. on LISP and Functional Programming, 1986.

[9] F.Nielson: Strictness analysis and denotational abstract interpretation (extended abstract), Proceedings from the 1987 ACM Conf. on Principles of Programming Languages, 1987. A full version is to appear in Information and Computation.

[10] F.Nielson: A formal type system for comparing partial evaluators, The Technical University of Denmark, 1987. To appear in proceedings from Workshop on Partial Evaluation and Mixed Computation, North Holland, 1988.

[11] H.R.Nielson, F.Nielson: Automatic binding time analysis for a typed λ-calculus, to appear in Proceedings from the 1988 ACM Conf. on Principles of Programming Languages, 1988.

[12] D.Schmidt: Static properties of partial reduction, Kansas State University, 1987. To appear in proceedings from Workshop on Partial Evaluation and Mixed Computation, North Holland, 1988.

[13] M.Schoenfinkel: Über die Bausteine der mathematischen Logik, Mathematische Annalen, vol.92, 1924.

[14] R.D.Tennent: Principles of Programming Languages, Prentice Hall, 1981.

[15] D.A.Turner: A new implementation technique for applicative languages, Software – Practice and Experience, vol.9, 1979.

[16] D.A.Turner: Another algorithm for bracket abstraction, The Journal of Symbolic Logic, vol.44, 1979.