Multi-Level Lambda-Calculi: an Algebraic Description

(Draft as of January 1996)

Flemming Nielson and Hanne Riis Nielson

DAIMI, Aarhus University, Denmark.

Abstract. Two-level λ-calculi have been heavily utilised for applications such as partial evaluation, abstract interpretation and code generation. Each of these applications pose different demands on the exact details of the two-level structure and the corresponding inference rules. We therefore formulate a number of existing systems in a common framework. This is done in such a way as to conceal those differences between the systems that are not essential for the multi-level ideas (like whether or not one restricts the domain of the type environment to the free identifiers of the expression) and thereby to reveal the deeper similarities and differences. In their most general guise the multi-level λ-calculi allow multi-level structures that are not restricted to (possibly finite) linear orders and thereby generalise previous treatments in the literature.

1 Introduction

The concept of two-level languages is at least a decade old [7, 4] and the concept of multi-level languages at least four years old [11]. In particular two-level languages have been used extensively in the development of partial evaluation [1, 3] and abstract interpretation [5, 8] but also in areas such as code generation [9] and processor placement [12].

One goal of this paper is to cast further light on the two-level λ-calculi that may be found in the literature. We will show that there is a high degree of *commonality* in the approach taken: there are a number of levels (e.g. binding-times) and relations between them. Also we will stress that there are major *differences* that to a large extent are *forced* by the characteristics of the application domains (be it partial evaluation, code generation, abstract interpretation, or processor placement). In our view it is important to understand this point, that the application domains place different demands on the formalisation, before it makes sense to compare formalisations with a view to identifying their relative virtues.

Another goal of this paper is to synthesise these considerations into a general description of multi-level languages. In this paper we solve the problem of generalising from a (possibly finite) linearly ordered structure to a much more general structure of levels. We thus pave the way for future work on further making the notions independent of the λ-calculus and on defining parameterised classes of multi-level λ-calculi.

Remark. Perhaps the most obvious generalisation of a notion of two levels is a (possibly finite) interval in $\mathbf{Z} \cup \{-\infty, \infty\}$ (with the elements corresponding to the levels [11, 1]). A somewhat more abstract possibility is to use a general partially ordered set (with the elements corresponding to the levels as is briefly discussed in [11]) although a Kripke-structure [6] (with the worlds corresponding to the levels) would fit just as well. This might suggest that the ultimate choice is to let the levels be given by a category because a partial order can indeed be viewed as a particularly simple category. We shall find it more appropriate[1] to use a many-sorted algebra with sorts corresponding to the levels and operators corresponding to the relationships between the levels; one reason is that it avoids the need for coding many-argument concepts as one-argument concepts using cartesian products, another is that it naturally allows different relationships between the levels for different syntactic categories, and yet a third reason is that it allows finer control over the relationship between the levels in that it does not necessarily impose transitivity.

2 Preliminaries

Programming languages are characterised by a number of syntactic categories and by a number of constructs for combining syntactic entities to new ones. Using the terminology of many-sorted algebras we shall represent the set of names of syntactic categories as a set of sorts and the methods as operators. To this end we begin by reviewing some concepts from many-sorted algebras [14, 2].

A many-sorted signature Σ over a set S of sorts consists of a set (also denoted Σ) of operators; each operator $\sigma \in \Sigma$ is assigned a rank, denoted $\mathsf{rank}(\sigma) \in S^* \times S$, designating the sequence of sorts of the arguments and the sort of the result; if $\mathsf{rank}(\sigma) = (s_1 \cdots s_n; s)$ we shall say that σ is an n-ary operator.

A Σ-algebra M consists of a (usually non-empty) set M_s (called the carrier) for each sort $s \in S$ and a total function $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \to M_s$ for each operator $\sigma \in \Sigma$ of rank $(s_1 \cdots s_n; s)$.

The free Σ-algebra $T(\Sigma)$ has as carrier $T(\Sigma)_s$ the set of terms of sort s that can be built using the operators of Σ; as operators it has the constructions of

[1] These approaches are not too dissimilar in their descriptive power and thus largely a matter of taste: a many-sorted algebra can be regarded as a cartesian category (with objects corresponding to sequences of sorts and operators corresponding to morphisms), and a cartesian category can be regarded as a many-sorted algebra (with sequences of sorts corresponding to objects and morphisms corresponding to operators).

new terms. In a similar way the free Σ-algebra $T(\Sigma, X)$ over X has as carrier $T(\Sigma, X)_s$ the set of terms of sort s that can be built using the operators of Σ and the identifiers in X where each identifier $x \in X$ has an associated sort, denoted $\mathsf{sort}(x)$; as operators it has the constructions of new terms. Another way to present this is to say that $T(\Sigma, X) = T(\Sigma \cup X)$ where the rank of x is given by $\mathsf{rank}(x) = (;\mathsf{sort}(x))$.

A homomorphism h from a Σ-algebra M_1 to a Σ-algebra M_2 consists of a sort-preserving mapping from the carriers of M_1 to those of M_2 such that for each operator $\sigma \in \Sigma$ and for all values $v_1, \cdots, v_n$ in M_1 of the required sorts, the equation $h(\sigma_{M_1}(v_1, \cdots, v_n)) = \sigma_{M_2}(h(v_1), \cdots, h(v_n))$ holds in M_2.

A derivor d from a signature Σ_1 over S to a signature Σ_2 over S is a mapping that sends an operator $\sigma \in \Sigma_1$ of rank $(s_1 \cdots s_n; s)$ to a term $d(\sigma) \in T(\Sigma_2, \{x_1, \cdots, x_n\})_s$ constructed from the operators of Σ_2 together with the identifiers $\{x_1, \cdots, x_n\}$ such that if each x_i has sort $\mathsf{sort}(x_i) = s_i$ then $d(\sigma)$ obeys the sorting rules and gives a term of sort s. (This definition can be made more general by allowing Σ_1 and Σ_2 to have different sets of sorts as is done in [2].)

We shall define a *uniform derivor* from a signature Σ_1 over S to a signature Σ_2 over S to be a rank-preserving partial mapping δ from Σ_1 to Σ_2 that is only allowed to be undefined on unary operators of Σ_1; it extends to a derivor (also denoted δ) by mapping $\sigma \in \Sigma_1$ of rank $(s_1 \cdots s_n; s)$ to $(\delta(\sigma))(x_1, \cdots, x_n)$ if $\delta(\sigma)$ is defined and to x_1 otherwise; note that for this derivor all $\delta(\sigma)$ contains at most one operator symbol. (We should point out that a uniform derivor is an instance of a signature morphism [14] whenever it happens to be a total mapping.)

3 A descriptive approach to multi-level lambda-calculi

In this section we shall define the syntax of the lambda-calculus and some common features of multi-level structures. The aim is to provide a small universe in which some of the different formalisations of multi-level languages found in the litterature can be explained as necessary variations over a theme.

λ-calculus. The simply typed λ-calculus $\boldsymbol{\lambda}$ is the programming language specified by the following data. The sorts (or syntactic categories) are Typ and Exp. The signature (or the set of type and expression forming constructs) Σ is given by:

$\rightarrow: (\mathsf{Typ}^2; \mathsf{Typ})$	$\mathtt{int} : (; \mathsf{Typ})$	$\mathtt{bool} : (; \mathsf{Typ})$
$c_i : (; \mathsf{Exp})$	$x_i : (; \mathsf{Exp})$	$\lambda x_i. : (\mathsf{Exp}; \mathsf{Exp})$
$\mathtt{@} : (\mathsf{Exp}^2; \mathsf{Exp})$	$\mathtt{if} : (\mathsf{Exp}^3; \mathsf{Exp})$	$\mathtt{fix} : (\mathsf{Exp}; \mathsf{Exp})$

for i ranging over some index set. There are two well-formedness judgements: $\vdash^{\mathsf{T}} t$ for the well-formedness of the type t and $A \vdash^{\mathsf{E}} e : t$ for the well-formedness of the expression e (yielding type t assuming free identifiers are typed according to the type environment A). The inductive definition of these well-formedness judgements is given by the following inference rule for $\vdash^{\mathsf{T}}$:

$$[\mathrm{ok}]\ \frac{}{\vdash^{\mathsf{T}} t}$$

(stating that all types are well-formed and where we regard an axiom as an inference rule with no premises) and for $\vdash^{\mathsf{E}}$:

$$[c_i]\ \frac{}{A \vdash^{\mathsf{E}} c_i : t} \text{ if } t = \mathsf{Type}(c_i) \qquad [x_i]\ \frac{}{A \vdash^{\mathsf{E}} x_i : t} \text{ if } t = A(x_i)$$

$$[\lambda x_i]\ \frac{A[x_i : t_i] \vdash^{\mathsf{E}} e : t}{A \vdash^{\mathsf{E}} \lambda x_i.e : t_i \to t} \qquad [\texttt{@}]\ \frac{A \vdash^{\mathsf{E}} e_0 : t_1 \to t_2 \qquad A \vdash^{\mathsf{E}} e_1 : t_1}{A \vdash^{\mathsf{E}} e_0 \texttt{@}\, e_1 : t_2}$$

$$[\texttt{fix}]\ \frac{A \vdash^{\mathsf{E}} e : t \to t}{A \vdash^{\mathsf{E}} \texttt{fix}\ e : t} \qquad [\texttt{if}]\ \frac{A \vdash^{\mathsf{E}} e_0 : \texttt{bool} \quad A \vdash^{\mathsf{E}} e_1 : t \quad A \vdash^{\mathsf{E}} e_2 : t}{A \vdash^{\mathsf{E}} \texttt{if}\ e_0\ e_1\ e_2 : t}$$

for some unspecified table Type giving the type of constants.

Note that this is just the algebraic presentation of the well-known simply typed λ-calculus: we have the two syntactic categories (represented by the sorts), we have the abstract syntax (represented by the signature), and we have the well-formedness judgements and the inference rules for their definition. We are stepping slightly outside the algebraic framework in allowing type environments, and operations upon these, even though there is no sort corresponding to type environments; this could very easily be rectified but at the price of a more cumbersome formalisation.

Remark. An alternative presentation $\boldsymbol{\lambda}'$ of the simply typed λ-calculus has the same sorts, the same signature, the same well-formedness judgements but other rules of inference. For $\vdash^{\mathsf{T}}$ it has:

$$[\texttt{int}]\ \frac{}{\vdash^{\mathsf{T}} \texttt{int}} \qquad [\texttt{bool}]\ \frac{}{\vdash^{\mathsf{T}} \texttt{bool}} \qquad [\to]\ \frac{\vdash^{\mathsf{T}} t_1 \qquad \vdash^{\mathsf{T}} t_2}{\vdash^{\mathsf{T}} t_1 \to t_2}$$

and for $\vdash^{\mathsf{E}}$ the rule $[\lambda x_i]$ is changed to:

$$[\lambda x_i]\ \frac{A[x_i : t_i] \vdash^{\mathsf{E}} e : t}{A \vdash^{\mathsf{E}} \lambda x_i.e : t_i \to t} \text{ if } \vdash^{\mathsf{T}} t_i$$

Since all types in $\boldsymbol{\lambda}'$ are well-formed just as in $\boldsymbol{\lambda}$ the two presentations are for all practical purposes equivalent. Consequently our development below must be sufficiently flexible that it can be based on $\boldsymbol{\lambda}$ as well as $\boldsymbol{\lambda}'$.

Multi-level structure. A multi-level structure B (for $\boldsymbol{\lambda}$) is characterised by the sorts Typ and Exp, a non-empty set W^B (also denoted B) of levels and an explicitly given

– $(W^B \times \{\mathsf{Typ}, \mathsf{Exp}\})$-sorted signature Ω^B

that is regarded as *additionally* containing the implicitly given operators $\iota^b_{s_1 \cdots s_n;s}$ of rank $((b, s_1) \cdots (b, s_n); (b, s))$ for all $b \in B$ and all $s_i, s \in \{\mathsf{Typ}, \mathsf{Exp}\}$. We shall allow to write Ω^B_e to indicate the subset of explicitly given operators (i.e. excluding those of form $\iota^b_{s_1 \cdots s_n;s}$). We shall write $|B|$ for the cardinality of W^B.

As we shall see the intention is that the implicit operators ι allow arbitrary inference rules as long as we stay at the same level but that whenever we change levels there must be an explicitly given operator that supports (or permits) this. We shall illustrate this with examples below.

Multi-level λ-calculus. A multi-level λ-calculus L over B (and $\boldsymbol{\lambda}$) is characterised by the sorts Typ and Exp, the multi-level structure B, the well-formedness judgements $\vdash^{\mathsf{T}}_b t$ and $A \vdash^{\mathsf{E}}_b e : t$ (where b ranges over B), and the following explicitly given information:

– a $\{\mathsf{Typ}, \mathsf{Exp}\}$-sorted signature Σ^L (defining the syntax of L), and
– a set R^L of named inference rules for the well-formedness judgements, and
– a uniform derivor $\delta : \Sigma^L \to \Sigma$ that is extended to map $\vdash^s_b$ to $\vdash^s$ and thereby may be used to map judgements and inferences of L to judgements and inferences of $\boldsymbol{\lambda}$ in a mostly compositional manner, and

such that each inference rule $\Delta \in R^L$ satisfies:

(i) it identifies an operator $\omega \in \Omega^B$ of rank $((b_1, s_1) \cdots (b_n, s_n); (b, s))$ such that the premises of Δ concern the well-formedness judgements $\vdash^{s_i}_{b_i}$ and the conclusion concerns the well-formedness judgement $\vdash^s_b$ and the only judgements $\vdash^{s'}_{b'}$ allowed in the side condition have $b' = b$; and
(ii) the rule $\delta(\Delta)$ is a permissible[2] rule in $\boldsymbol{\lambda}$.

3.1 Example: code generation

We shall now see that the restriction of the two-level λ-calculus of [11] to $\boldsymbol{\lambda}$ (summarised in Appendix A.1) is an instance of the present framework. To this end we define the two-level language $L = L_{cg}$.

[2] We say that a rule Δ is a permissible rule for a rule set R whenever the set of provable judgements using R equals the set of provable judgements using $R \cup \{\Delta\}$. More restrictive demands on a rule might be that it is a derived rule or even that it is an existing rule in R.

Two-level structure. Let B contain the two levels c (for compile-time) and r (for run-time). The signature Ω^B then has the following explicitly given operators:

- *UP*: $((r, \mathsf{Typ}); (c, \mathsf{Typ}))$
- *up*: $((r, \mathsf{Exp}); (c, \mathsf{Exp}))$
- *dn*: $((c, \mathsf{Exp}); (r, \mathsf{Exp}))$

The operator *UP* indicates that run-time types can be embedded in compile-time types thereby imposing the ordering that r is "less than" c. The operator *up* indicates that values of run-time expressions (i.e. code) can be manipulated at compile-time and the operator *dn* that values of compile-time expressions can be used at run-time.

Two-level λ-calculus. The signature Σ^L is given by:

$$\begin{array}{lll}
\rightarrow^c, \rightarrow^r : (\mathsf{Typ}^2; \mathsf{Typ}) & \mathtt{int}^c, \mathtt{int}^r : (; \mathsf{Typ}) & \mathtt{bool}^c, \mathtt{bool}^r : (; \mathsf{Typ}) \\
c_i^c, c_i^r : (; \mathsf{Exp}) & x_i : (; \mathsf{Exp}) & \lambda^c x_i., \lambda^r x_i. : (\mathsf{Exp}; \mathsf{Exp}) \\
\mathtt{@}^c, \mathtt{@}^r : (\mathsf{Exp}^2; \mathsf{Exp}) & \mathtt{if}^c, \mathtt{if}^r : (\mathsf{Exp}^3; \mathsf{Exp}) & \mathtt{fix}^c, \mathtt{fix}^r : (\mathsf{Exp}; \mathsf{Exp})
\end{array}$$

Note that we have two copies of every operator of Σ (except identifiers that can be viewed as place-holders). Annotations with r corresponds to the underlining notation used in [11] and annotations with c to the absence of underlinings.

For *types* the well-formedness rules include two copies of the well-formedness rules of $\boldsymbol{\lambda}'$ (one for $b = c$ and one for $b = r$):

$$[\iota^b]\ \frac{}{\vdash_b^{\top} \mathtt{int}^b} \qquad [\iota^b]\ \frac{}{\vdash_b^{\top} \mathtt{bool}^b} \qquad [\iota^b]\ \frac{\vdash_b^{\top} t_1 \quad \vdash_b^{\top} t_2}{\vdash_b^{\top} t_1 \rightarrow^b t_2}$$

On top of this we have a bridging rule corresponding to the operator *UP* of the two-level structure:

$$[UP]\ \frac{\vdash_r^{\top} t_1 \rightarrow^r t_2}{\vdash_c^{\top} t_1 \rightarrow^r t_2}$$

allowing us to transfer run-time function spaces to compile-time. It is trivial to verify that we have given the correct treatment of types:

Fact 1. $\vdash_b^{\top} t$ *if and only if* $\vdash t : b$ *(in Appendix A.1).*

For *expressions* we have two slightly modified copies of the well-formedness rules of $\boldsymbol{\lambda}'$ (one for $b = c$ and one for $b = r$). To capture the formulation of [11] we shall let the type environment A associate a level b and a type t with each identifier x_i:

$$[\iota^b]\ \frac{}{A \vdash^{\mathsf{E}}_{b} c_i^b : t}\ \text{if } t = \mathsf{Type}(c_i^b) \wedge \vdash^{\mathsf{T}}_{b} t \quad [\iota^b]\ \frac{}{A \vdash^{\mathsf{E}}_{b} x_i : t}\ \text{if } t = A(x_i^b) \wedge \vdash^{\mathsf{T}}_{b} t$$

$$[\iota^b]\ \frac{A[x_i^b : t_i] \vdash^{\mathsf{E}}_{b} e : t}{A \vdash^{\mathsf{E}}_{b} \lambda^b x_i.e : t_i \rightarrow^b t}\ \text{if } \vdash^{\mathsf{T}}_{b} t_i \qquad [\iota^b]\ \frac{A \vdash^{\mathsf{E}}_{b} e_0 : t_1 \rightarrow^b t_2 \qquad A \vdash^{\mathsf{E}}_{b} e_1 : t_1}{A \vdash^{\mathsf{E}}_{b} e_0 \,\mathtt{@}^b\, e_1 : t_2}$$

$$[\iota^b]\ \frac{A \vdash^{\mathsf{E}}_{b} e : t \rightarrow^b t}{A \vdash^{\mathsf{E}}_{b} \mathtt{fix}^b\ e : t} \qquad [\iota^b]\ \frac{A \vdash^{\mathsf{E}}_{b} e_0 : \mathtt{bool}^b \quad A \vdash^{\mathsf{E}}_{b} e_1 : t \quad A \vdash^{\mathsf{E}}_{b} e_2 : t}{A \vdash^{\mathsf{E}}_{b} \mathtt{if}^b\ e_0\ e_1\ e_2 : t}$$

where as before[3] we leave the table Type unspecified. On top of this we have two bridging rules corresponding to the operators *up* and *dn* of the two-level structure:

$$[dn]\ \frac{A' \vdash^{\mathsf{E}}_{c} e : t}{A \vdash^{\mathsf{E}}_{r} e : t}\ \text{if } \vdash^{\mathsf{T}}_{r} t \wedge \mathsf{gr}(A') \subseteq \mathsf{gr}(A)$$

$$[up]\ \frac{A' \vdash^{\mathsf{E}}_{r} e : t}{A \vdash^{\mathsf{E}}_{c} e : t}\ \text{if } \vdash^{\mathsf{T}}_{c} t \wedge \mathsf{gr}(A') \subseteq \mathsf{gr}(A) \wedge \forall (x_i^{b'} : t') \in \mathsf{gr}(A') : (b' = c \wedge \vdash^{\mathsf{T}}_{c} t')$$

where $\mathsf{gr}(A) = \{(x_i^b : t) \mid A(x_i^b) = t\}$ is the *graph* of A. It is trivial to establish the following relationship between the typing judgements:

Fact 2. *$A \vdash^{\mathsf{E}}_{b} e : t$ implies $\vdash^{\mathsf{T}}_{b} t$.*

To show that we have given the correct treatment for expressions we define a mapping upon the type environments of Appendix A.1:

$$\langle \cdots [x_i : t : b] \cdots \rangle = \langle \cdots \rangle [x_i^b : t] \langle \cdots \rangle$$

and we then prove:

Lemma 3. *$\langle A \rangle \vdash^{\mathsf{E}}_{b} e : t$ if and only if $A \vdash e : t : b$ (in Appendix A.1).*

Finally we have to specify a *uniform derivor* δ from the two-level λ-calculus into $\boldsymbol{\lambda}$: it simply removes all annotations. It is fairly straightforward to check that the conditions on the multi-level λ-calculus are satisfied. The same story goes for letting the uniform derivor map into $\boldsymbol{\lambda}'$. It is instructive to point out that although we modelled the two-level λ-calculus after $\boldsymbol{\lambda}'$ our notion of two-level language is flexible enough to let the derivor map it back to $\boldsymbol{\lambda}$ as well as $\boldsymbol{\lambda}'$.

[3] Actually there is a small subtlety here concerning the $A[x_i^b : t_i]$ notation: if A already contains $[x_i^{b'} : t_i']$ for $b' \neq b$ will the update then remove the entry for $x_i^{b'}$ or not? In line with [11] we shall assume that the entry *is* removed, although it would be feasible to take the other approach and then perhaps replace the operators $x_i \in \Sigma$ with $x_i^b \in \Sigma$.

3.2 Example: partial evaluation

We shall now see that the restriction of the binding time analysis of [3] to $\boldsymbol{\lambda}$ (summarised in Appendix A.2) is an instance of the present framework. To this end we define the two-level language $L = L_{pe}$.

Two-level structure. Let B contain the two levels D (for dynamic) and S (for static). The signature Ω^B then has the following explicitly given operators:

- DN: $((D, \mathsf{Typ}); (S, \mathsf{Typ}))$
- dn: $((D, \mathsf{Exp}); (S, \mathsf{Exp}))$
- up, $coer$: $((S, \mathsf{Exp}); (D, \mathsf{Exp}))$

The operator DN indicates that dynamic types can be embedded in static types; usually this is reflected by imposing an ordering $S \leq D$ saying that S computations take place "before" D computations[4]. The operators dn and up reflect that expressions at the two levels can be mixed much as in Subsection 3.1 and the presence of $coer$ reflects that some form of coercion of static values to dynamic values can take place.

Two-level λ-calculus. We use the following signature Σ^L:

$$\begin{array}{lll}
\rightarrow^D, \rightarrow^S : (\mathsf{Typ}^2; \mathsf{Typ}) & \mathtt{int}^D, \mathtt{int}^S : (; \mathsf{Typ}) & \mathtt{bool}^D, \mathtt{bool}^S : (; \mathsf{Typ}) \\
c_i^D, c_i^S : (; \mathsf{Exp}) & x_i : (; \mathsf{Exp}) & \lambda^D x_i., \lambda^S x_i. : (\mathsf{Exp}; \mathsf{Exp}) \\
\mathtt{@}^D, \mathtt{@}^S : (\mathsf{Exp}^2; \mathsf{Exp}) & \mathtt{if}^D, \mathtt{if}^S : (\mathsf{Exp}^3; \mathsf{Exp}) & \mathtt{fix}^S : (\mathsf{Exp}; \mathsf{Exp})
\end{array}$$

This is very similar to Subsection 3.1 except that (following [3]) there is no $\mathtt{fix}^D$, i.e. all fix point computations will be static.

For *types* we first introduce the following well-formedness rules:

$$[\iota^b]\ \frac{}{\vdash_b^{\top} \mathtt{int}^b} \qquad [\iota^b]\ \frac{}{\vdash_b^{\top} \mathtt{bool}^b} \qquad [\iota^b]\ \frac{}{\vdash_b^{\top} t_1 \rightarrow^b t_2}$$

(where b ranges over $\{S, D\}$). Note that the rule for $t_1 \rightarrow^b t_2$ has no premises! Then we have the following bridging rule corresponding to the operator DN:

$$[DN]\ \frac{\vdash_D^{\top} t}{\vdash_S^{\top} t}$$

[4] Intuitively, the level D corresponds to the level r of Subsection 3.1 and similarly the level S corresponds to the level c. The ordering imposed on D and S above will then be the dual of the ordering imposed on c and r in Subsection 3.1. This is analogous to the dual orderings used in data flow analysis and in abstract interpretation.

allowing us to use any dynamic type as a static type. One can then prove that we have given the correct treatment of types:

Fact 4. $\vdash_b^{\mathsf{T}} t$ *if and only if* $b \leq \mathsf{top}(t)$ *(in Appendix A.2).*

For *expressions* we first introduce the following slightly modified copies of rules from $\boldsymbol{\lambda}'$:

$$[\iota^b]\ \frac{}{A \vdash_b^{\mathsf{E}} c_i^b : t} \text{ if } t = \mathsf{Type}(c_i^b) \ \wedge\ \vdash_b^{\mathsf{T}} t \qquad [\iota^b]\ \frac{}{A \vdash_b^{\mathsf{E}} x_i : t} \text{ if } t = A(x_i) \ \wedge\ \vdash_b^{\mathsf{T}} t$$

$$[\iota^b]\ \frac{A[x_i : t_i] \vdash_b^{\mathsf{E}} e : t}{A \vdash_b^{\mathsf{E}} \lambda^b x_i.e : t_i \rightarrow^b t} \text{ if } \vdash_b^{\mathsf{T}} t_i \qquad [\iota^b]\ \frac{A \vdash_b^{\mathsf{E}} e_0 : t_1 \rightarrow^b t_2 \qquad A \vdash_b^{\mathsf{E}} e_1 : t_1}{A \vdash_b^{\mathsf{E}} e_0 \texttt{@}^b\, e_1 : t_2} \text{ if } \vdash_b^{\mathsf{T}} t_2$$

$$[\iota^b]\ \frac{A \vdash_S^{\mathsf{E}} e : t \rightarrow^b t}{A \vdash_S^{\mathsf{E}} \texttt{fix}^S\ e : t} \qquad [\iota^b]\ \frac{A \vdash_b^{\mathsf{E}} e_0 : \texttt{bool}^b \quad A \vdash_b^{\mathsf{E}} e_1 : t \quad A \vdash_b^{\mathsf{E}} e_2 : t}{A \vdash_b^{\mathsf{E}} \texttt{if}^b\ e_0\ e_1\ e_2 : t}$$

(where b ranges over $\{S, D\}$). Note that compared with Subsection 3.1 we have not extended the entries in A with information about the level. In addition we have the following bridging rules corresponding to the operators *up*, *dn* and *coer*:

$$[up]\ \frac{A \vdash_S^{\mathsf{E}} e : t}{A \vdash_D^{\mathsf{E}} e : t} \text{ if } \vdash_D^{\mathsf{T}} t \qquad [dn]\ \frac{A \vdash_D^{\mathsf{E}} e : t}{A \vdash_S^{\mathsf{E}} e : t}$$

$$[coer]\ \frac{A \vdash_S^{\mathsf{E}} e : \texttt{int}^S}{A \vdash_D^{\mathsf{E}} e : \texttt{int}^D} \qquad [coer]\ \frac{A \vdash_S^{\mathsf{E}} e : \texttt{bool}^S}{A \vdash_D^{\mathsf{E}} e : \texttt{bool}^D}$$

Note that the rule [*coer*] has no counterpart in Subsection 3.1. It is trivial to establish the following relationship between the typing judgements:

Fact 5. $A \vdash_b^{\mathsf{E}} e : t$ *implies* $\vdash_b^{\mathsf{T}} t$.

To show that we have given the correct treatment for expressions we prove:

Lemma 6. $A \vdash_b^{\mathsf{E}} e : t$ *if and only if* $A \vdash e : t$ *and* $b \leq \mathsf{top}(t)$ *(in Appendix A.2).*

Finally we have to specify a uniform derivor δ from the two-level λ-calculus into $\boldsymbol{\lambda}$: it simply removes all annotations. It is fairly straightforward to check that the conditions on the multi-level λ-calculus are satisfied.

Remark. In the above rule for $t_1 \rightarrow^b t_2$ it is *not* required that the subtypes t_1 and t_2 are well-formed. So using the system of [3] one can in fact prove

$$\emptyset \vdash \lambda^D x.x : (\texttt{int}^S \rightarrow^D \texttt{int}^S) \rightarrow^D (\texttt{int}^S \rightarrow^D \texttt{int}^S) \qquad (*)$$

One may argue that this is unfortunate since traditional partial evaluators cannot exploit this information. However, we can easily rectify this in our setting: replace the above rule for $t_1 \rightarrow^b t_2$ with

$$\frac{\vdash_b^{\mathsf{T}} t_1 \qquad \vdash_b^{\mathsf{T}} t_2}{\vdash_b^{\mathsf{T}} t_1 \rightarrow^b t_2}$$

thus bringing the system closer to that of Subsection 3.1. As a consequence we can remove the side condition $\vdash_b^{\mathsf{T}} t_2$ from the rule for application since well-formedness of t_2 now can be deduced from the well-formedness of $t_1 \rightarrow^b t_2$. Note that with these changes (*) is no longer derivable. We call this new system L'_{pe}.

3.3 Example: multi-level partial evaluation

We shall now see that the restriction of the multi-level binding time analysis of [1] to $\boldsymbol{\lambda}$ (summarised in Appendix A.3) is an instance of the present framework. To this end we define the two-level language $L = L_{mp}$.

Multi-level structure. Let B contain the levels $0, 1, \cdots, \mathsf{max}$ where intuitively 0 stands for static and $1, \cdots, \mathsf{max}$ for different levels of dynamic. The signature Ω^B then has the following explicitly given operators:

- $DN_b^{b'}$: $((b + b', \mathsf{Typ}); (b, \mathsf{Typ}))$ for $0 \leq b \leq b + b' \leq \mathsf{max}$
- $dn_b^{b'}$: $((b + b', \mathsf{Exp}); (b, \mathsf{Exp}))$ for $0 \leq b \leq b + b' \leq \mathsf{max}$
- $up_b^{b'}$, $lift_b^{b'}$: $((b, \mathsf{Exp}); (b + b', \mathsf{Exp}))$ for $0 \leq b \leq b + b' \leq \mathsf{max}$

Thus $DN_b^{b'}$ allows us to embed types at level $b + b'$ at the lower level b; this imposes the ordering that $b \leq b + b'$ much as in Subsection 3.2. The operators $dn_b^{b'}$ and $up_b^{b'}$ reflect that expressions on the various levels can be mixed and the presence of $lift_b^{b'}$ reflects that some form of lifting of values at level b to level $b + b'$ can be performed.

Note that if we were to restrict b' to be 1 we would only be able to move between adjacent levels in B although we could of course repeat such moves.

Multi-level λ-calculus. We use the following signature Σ^L where $b \in \{0, 1, \cdots, \mathsf{max}\}$:

$\rightarrow^b : (\mathsf{Typ}^2; \mathsf{Typ})$	$\mathtt{int}^b : (; \mathsf{Typ})$	$\mathtt{bool}^b : (; \mathsf{Typ})$
$c_i^b : (; \mathsf{Exp})$	$x_i : (; \mathsf{Exp})$	$\lambda^b x_i. : (\mathsf{Exp}; \mathsf{Exp})$
$\mathtt{@}^b : (\mathsf{Exp}^2; \mathsf{Exp})$	$\mathtt{if}^b : (\mathsf{Exp}^3; \mathsf{Exp})$	$\mathtt{fix}^0 : (\mathsf{Exp}; \mathsf{Exp})$

$\mathtt{lift}_b^{b'} : (\mathsf{Exp}; \mathsf{Exp})$ for $0 \leq b \leq b + b' \leq \mathsf{max}$

As in Subsection 3.2 all fix point computations are required to be static[5], i.e. at level 0. Note that in addition to the annotations on the operators of $\boldsymbol{\lambda}$ we also have the new operators $\mathtt{lift}_b^{b'}$.

[5] In [1] recursive computations are specified implicitly.

For *types* we first introduce the following well-formedness rules:

$$[\iota^b]\ \frac{}{\vdash^{\mathsf{T}}_{b} \mathtt{int}^b} \qquad [\iota^b]\ \frac{}{\vdash^{\mathsf{T}}_{b} \mathtt{bool}^b} \qquad [\iota^b]\ \frac{\vdash^{\mathsf{T}}_{b} t_1 \quad \vdash^{\mathsf{T}}_{b} t_2}{\vdash^{\mathsf{T}}_{b} t_1 \rightarrow^b t_2}$$

(where b ranges over $\{0, 1, \cdots, \mathsf{max}\}$). Then we have the following bridging rules corresponding to the operator $DN_b^{b'}$:

$$[DN_b^{b'}]\ \frac{\vdash^{\mathsf{T}}_{b+b'} t}{\vdash^{\mathsf{T}}_{b} t}$$

allowing us to use any type at level $b + b'$ at the lower level b. One can then prove that we have given the correct treatment of types:

Fact 7. $\vdash^{\mathsf{T}}_{b} t$ *if and only if* $\| t \| \geq b$ *(in Appendix A.3).*

For *expressions* we first introduce the following slightly modified copies of $\boldsymbol{\lambda}'$:

$$[\iota^b]\ \frac{}{A \vdash^{\mathsf{E}}_{b} c_i^b : t} \text{ if } t = \mathsf{Type}(c_i^b) \ \wedge\ \vdash^{\mathsf{T}}_{b} t \quad [\iota^b]\ \frac{}{A \vdash^{\mathsf{E}}_{b} x_i : t} \text{ if } t = A(x_i) \ \wedge\ \vdash^{\mathsf{T}}_{b} t$$

$$[\iota^b]\ \frac{A[x_i : t_i] \vdash^{\mathsf{E}}_{b} e : t}{A \vdash^{\mathsf{E}}_{b} \lambda^b x_i.e : t_i \rightarrow^b t} \text{ if } \vdash^{\mathsf{T}}_{b} t_i \qquad [\iota^b]\ \frac{A \vdash^{\mathsf{E}}_{b} e_0 : t_1 \rightarrow^b t_2 \qquad A \vdash^{\mathsf{E}}_{b} e_1 : t_1}{A \vdash^{\mathsf{E}}_{b} e_0 \mathtt{@}^b\, e_1 : t_2}$$

$$[\iota^b]\ \frac{A \vdash^{\mathsf{E}}_{0} e : t \rightarrow^b t}{A \vdash^{\mathsf{E}}_{0} \mathtt{fix}^0\ e : t} \qquad [\iota^b]\ \frac{A \vdash^{\mathsf{E}}_{b} e_0 : \mathtt{bool}^b \quad A \vdash^{\mathsf{E}}_{b} e_1 : t \quad A \vdash^{\mathsf{E}}_{b} e_2 : t}{A \vdash^{\mathsf{E}}_{b} \mathtt{if}^b\ e_0\ e_1\ e_2 : t}$$

(where b ranges over $\{0, 1, \cdots, \mathsf{max}\}$). In addition we have the following bridging rules corresponding to the operators $up_b^{b'}$, $dn_b^{b'}$ and $lift_b^{b'}$:

$$[up_b^{b'}]\ \frac{A \vdash^{\mathsf{E}}_{b} e : t}{A \vdash^{\mathsf{E}}_{b+b'} e : t} \text{ if } \vdash^{\mathsf{T}}_{b+b'} t \qquad [dn_b^{b'}]\ \frac{A \vdash^{\mathsf{E}}_{b+b'} e : t}{A \vdash^{\mathsf{E}}_{b} e : t}$$

$$[lift_b^{b'}]\ \frac{A \vdash^{\mathsf{E}}_{b} e : \mathtt{int}^b}{A \vdash^{\mathsf{E}}_{b+b'} \mathtt{lift}_b^{b'}\ e : \mathtt{int}^{b+b'}} \quad [lift_b^{b'}]\ \frac{A \vdash^{\mathsf{E}}_{b} e : \mathtt{bool}^b}{A \vdash^{\mathsf{E}}_{b+b'} \mathtt{lift}_b^{b'}\ e : \mathtt{bool}^{b+b'}}$$

It is trivial to establish the following relationship between the typing judgements:

Fact 8. $A \vdash^{\mathsf{E}}_{b} e : t$ *implies* $\vdash^{\mathsf{T}}_{b} t$.

To show that we have given the correct treatment for expressions we prove:

Lemma 9. $A \vdash^{\mathsf{E}}_{b} e : t$ *if and only if* $A \vdash e : t$ *and* $\| t \| \geq b$ *(in Appendix A.3).*

Finally we have to specify a uniform derivor δ from the multi-level λ-calculus into $\boldsymbol{\lambda}$: it simply removes all annotations and all occurrences of $\mathtt{lift}_b^{b'}$. It is fairly straightforward to check that the conditions on the multi-level λ-calculus are satisfied.

3.4 Example: abstract interpretation

We shall now see that the two-level language TML[dt,dt] of [5] can be seen as an instance of the present framework. However, as our current framework does not directly support combinator introduction we shall prefer to consider a version of [5] where the combinators are replaced by λ-expressions; consequently it will be instructive to think only of forward program analyses. Given these considerations we can define the two-level language $L = L_{ai}$ as follows.

Two-level structure. The two-level structure B has the two levels d (for domain) and l (for lattice). The signature Ω^B has the following explicitly given operators:

- *UP*: $((l, \mathsf{Typ}); (d, \mathsf{Typ}))$
- *DN*: $((d, \mathsf{Typ}), (l, \mathsf{Typ}); (l, \mathsf{Typ}))$
- *up*: $((l, \mathsf{Exp}); (d, \mathsf{Exp}))$
- *dn*: $((d, \mathsf{Exp}); (l, \mathsf{Exp}))$

Here *UP* reflects that a lattice is a domain, and *DN* reflects that a domain and a lattice in certain cases can be put together and produce a lattice. The operations *up* and *dn* reflect that expressions denoting elements of domains and lattices can be mixed much as compile-time/run-time and static/dynamic expressions could in Subsections 3.1 and 3.2.

Two-level λ-calculus. We shall basically use the same signature Σ^L as in Subsection 3.1:

$$\begin{array}{lll}
\rightarrow^d, \rightarrow^l : (\mathsf{Typ}^2; \mathsf{Typ}) & \mathtt{int}^d, \mathtt{int}^l : (; \mathsf{Typ}) & \mathtt{bool}^d, \mathtt{bool}^l : (; \mathsf{Typ}) \\
c_i^d, c_i^l : (; \mathsf{Exp}) & x_i : (; \mathsf{Exp}) & \lambda^d x_i., \lambda^l x_i. : (\mathsf{Exp}; \mathsf{Exp}) \\
\mathtt{@}^d, \mathtt{@}^l : (\mathsf{Exp}^2; \mathsf{Exp}) & \mathtt{if}^d, \mathtt{if}^l : (\mathsf{Exp}^3; \mathsf{Exp}) & \mathtt{fix}^d, \mathtt{fix}^l : (\mathsf{Exp}; \mathsf{Exp})
\end{array}$$

For *types* the well-formedness rules include two copies of the well-formedness rules of $\boldsymbol{\lambda}'$ as was the case in Subsection 3.1:

$$[\iota^b]\ \frac{}{\vdash^{\mathsf{T}}_b \mathtt{int}^b} \qquad [\iota^b]\ \frac{}{\vdash^{\mathsf{T}}_b \mathtt{bool}^b} \qquad [\iota^b]\ \frac{\vdash^{\mathsf{T}}_b t_1 \quad \vdash^{\mathsf{T}}_b t_2}{\vdash^{\mathsf{T}}_b t_1 \rightarrow^b t_2}$$

(where b ranges over $\{l, d\}$). On top of this we have the bridging rule

$$[UP]\ \frac{\vdash^{\mathsf{T}}_l t}{\vdash^{\mathsf{T}}_d t}$$

which corresponds to the one in Subsection 3.2 and is somewhat more general than the one in Subsection 3.1; also we have an additional bridging rule

$$[DN]\ \frac{\vdash_d^{\mathsf{T}} t_1 \qquad \vdash_l^{\mathsf{T}} t_2}{\vdash_l^{\mathsf{T}} t_1 \rightarrow^d t_2}$$

that has no counterpart in Subsections 3.1 and 3.2. It is straightforward to show that $\vdash_d^{\mathsf{T}} t'$ holds if and only if $\mathbf{dt}(t')$ holds in [5], and that $\vdash_l^{\mathsf{T}} t'$ holds if and only if $\mathbf{lt}(t')$ holds in [5].

For *expressions* we have two slightly modified copies of the well-formedness rules of $\boldsymbol{\lambda}'$:

$$[\iota^b]\ \frac{}{A \vdash_b^{\mathsf{E}} c_i^b : t} \text{ if } t = \mathsf{Type}(c_i^b) \ \wedge \ \vdash_b^{\mathsf{T}} t \qquad [\iota^b]\ \frac{}{A \vdash_b^{\mathsf{E}} x_i : t} \text{ if } t = A(x_i^b) \ \wedge \ \vdash_b^{\mathsf{T}} t$$

$$[\iota^b]\ \frac{A[x_i^b : t_i] \vdash_b^{\mathsf{E}} e : t}{A \vdash_b^{\mathsf{E}} \lambda^b x_i.e : t_i \rightarrow^b t} \text{ if } \vdash_b^{\mathsf{T}} t_i \qquad [\iota^b]\ \frac{A \vdash_b^{\mathsf{E}} e_0 : t_1 \rightarrow^b t_2 \qquad A \vdash_b^{\mathsf{E}} e_1 : t_1}{A \vdash_b^{\mathsf{E}} e_0 \, \texttt{@}^b \, e_1 : t_2}$$

$$[\iota^b]\ \frac{A \vdash_b^{\mathsf{E}} e : t \rightarrow^b t}{A \vdash_b^{\mathsf{E}} \texttt{fix}^b \ e : t} \qquad [\iota^b]\ \frac{A \vdash_b^{\mathsf{E}} e_0 : \texttt{bool}^b \quad A \vdash_b^{\mathsf{E}} e_1 : t \quad A \vdash_b^{\mathsf{E}} e_2 : t}{A \vdash_b^{\mathsf{E}} \texttt{if}^b \ e_0 \ e_1 \ e_2 : t}$$

(where b ranges over $\{l, d\}$). On top of this we have the two bridging rules

$$[dn]\ \frac{A \vdash_d^{\mathsf{E}} e : t}{A \vdash_l^{\mathsf{E}} e : t} \text{ if } \vdash_l^{\mathsf{T}} t$$

$$[up]\ \frac{A \vdash_l^{\mathsf{E}} e : t}{A \vdash_d^{\mathsf{E}} e : t}$$

Finally we have to specify a uniform derivor δ: as in the previous examples it simply removes all annotations. It is fairly straightforward to check that the conditions on the multi-level λ-calculus are satisfied.

4 Conclusion

In this paper we have cast further light on some of the multi-level languages reported in the literature. This has had the effect of highlighting the essential differences as well as similarities and to pinpoint design decisions in existing calculi that should perhaps be reconsidered.

Generalisations of this work (already begun) would include dealing with arbitrary programming languages that need not be based on the λ-calculus and that need not be typed. Indeed they may have many more syntactic categories (for example declarations and statements) and more advanced typing constructs (polymorphism of one kind or the other).

In another direction the descriptive approach of the present paper should be complemented with a prescriptive approach as in [11]. This prescriptive approach should be more flexible than the one of [11] but is unlikely ever to be as flexible as a descriptive approach: it is like approximating a property from the below as well as the above (using a maxim from abstract interpretation). This work (already begun) is likely to focus on the λ-calculus rather than general multi-level languages as may be possible for the descriptive approach.

Acknowledgements. Thanks to Martin Berger, Dirk Dussart, and Jesper Jørgensen for fruitful discussions. This work was supported in part by the DART project (The Danish Research Councils) and the LOMAPS project (ESPRIT Basic Research).

References

1. R. Glück and J. Jørgensen: Efficient Multi-level Generating Extensions for Program Specialization. *PLILP'95*, Springer Lecture Notes in Computer Science, vol. 982: pp. 259–278, 1995.
2. J. A. Goguen and J. W. Thatcher and E. G. Wagner: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. *Current Trends in Programming Methodology*, vol. 4, (R. T. Yeh, editor), Prentice-Hall, 1978.
3. F. Henglein and C. Mossin: Polymorphic Binding-Time Analysis. *ESOP'94*, Springer Lecture Notes in Computer Science, vol. 788: pp. 287–301, 1994.
4. N. D. Jones and P. Sestoft and H. Søndergaard: An Experiment in Partial Evaluation: the Generation of a Compiler Generator. *Rewriting Techniques and Applications*, Springer Lecture Notes in Computer Science, vol. 202: pp. 124–140, 1985.
5. N. D. Jones and F. Nielson: Abstract Interpretation: a Semantics-Based Tool for Program Analysis. *Handbook of Logic in Computer Science*, vol. 4: 527–636, Oxford University Press, 1995.
6. J. C. Mitchell: Type Systems for Programming Languages. *Handbook of Theoretical Computer Science: Formal Models and Semantics*, vol. B: 365–458, Elsevier Science Publishers (and MIT Press), 1990.
7. F. Nielson: *Abstract Interpretation using Domain Theory.* PhD thesis, University of Edinburgh, Scotland, 1984.
8. F. Nielson: Two-Level Semantics and Abstract Interpretation. *Theoretical Computer Science — Fundamental Studies*, 69: 117–242, 1989.
9. F. Nielson and H. R. Nielson: Two-level semantics and code generation. *Theoretical Computer Science*, 56(1): 59–133, 1988.
10. H. R. Nielson and F. Nielson: Automatic Binding Time Analysis for a Typed λ-calculus. *Science of Computer Programming*, 10: 139–176, 1988.
11. F. Nielson and H. R. Nielson: *Two-Level Functional Languages.* Vol. 34 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1992.
12. F. Nielson and H. R. Nielson: Forced Transformations of Occam Programs. *Information and Software Technology*, 34(2): 91–96, 1992.

13. C. Strachey: The Varieties of Programming Languages. Technical Monograph PRG-10, Programming Research Group, University of Oxford, 1973.

14. M. Wirsing: Algebraic Specification. *Handbook of Theoretical Computer Science: Formal Models and Semantics*, vol. B: 675–788, Elsevier (and MIT Press), 1990.

A Subsets of existing systems

A.1 Code generation: [11]

In this subsection we summarise the binding time analysis of [11] (excluding product types and list types).

For types [11] defines a predicate $\vdash t : b$:

$$\frac{}{\vdash \texttt{int}^b : b} \qquad \frac{}{\vdash \texttt{bool}^b : b} \qquad \frac{\vdash t_1 : b \qquad \vdash t_2 : b}{\vdash t_1 \rightarrow^b t_2 : b} \qquad \frac{\vdash t_1 \rightarrow^r t_2 : r}{\vdash t_1 \rightarrow^r t_2 : c}$$

For expressions the typing rules have the form $A \vdash e : t : b$ and are defined by:

$$\frac{}{A \vdash c_i^b : t : b} \text{ if } t = \mathsf{Type}(c_i^b) \wedge \vdash t : b \qquad \frac{}{A \vdash x_i : t : b} \text{ if } (t : b) = A(x_i) \wedge \vdash t : b$$

$$\frac{A[x_i : (t_i : b)] \vdash e : t : b}{A \vdash \lambda^b x_i . e : t_i \rightarrow^b t : b} \text{ if } \vdash t_i : b \qquad \frac{A \vdash e_0 : t_1 \rightarrow^b t_2 : b \qquad A \vdash e_1 : t_1 : b}{A \vdash e_0 @^b e_1 : t_2 : b}$$

$$\frac{A \vdash e : t \rightarrow^b t : b}{A \vdash \texttt{fix}^b\ e : t : b} \qquad \frac{A \vdash e_0 : \texttt{bool}^b : b \qquad A \vdash e_1 : t : b \qquad A \vdash e_2 : t : b}{A \vdash \texttt{if}^b\ e_0\ e_1\ e_2 : t : b}$$

$$\frac{A \vdash e : t : c}{A \vdash e : t : r} \text{ if } \vdash t : r$$

$$\frac{A' \vdash e : t : r}{A \vdash e : t : c} \text{ if } \vdash t : c \ \wedge\ \mathsf{gr}(A') = \{(x_i : t' : b') \in \mathsf{gr}(A) \mid b' = c \wedge \vdash t' : c\}$$

A.2 Partial evaluation: [3]

In this subsection we present a restriction of the binding time analysis of [3] to the lambda calculus of the present paper. Compared with [3] we do not incorporate the qualified types (including polymorphism and constraints on binding times).

First define $\texttt{top}(t)$ to be the annotation at the top level of t, i.e. $\texttt{top}(\texttt{int}^b) = b$, $\texttt{top}(\texttt{bool}^b) = b$, and $\texttt{top}(t_1 \rightarrow^b t_2) = b$; in [3] one writes t^b to indicate that $\texttt{top}(t) = b$. Then the inference system for expressions is:

$$\frac{}{A \vdash c_i^b : t} \text{ if } t = \mathsf{Type}(c_i^b) \qquad \frac{}{A \vdash x_i : t} \text{ if } t = A(x_i)$$

$$\frac{A[x_i : t_i] \vdash e : t}{A \vdash \lambda^b x_i.e : t_i \rightarrow^b t} \text{ if } b \leq \mathsf{top}(t_i) \ \wedge \ b \leq \mathsf{top}(t)$$

$$\frac{A \vdash e_0 : t_1 \rightarrow^b t_2 \qquad A \vdash e_1 : t_1}{A \vdash e_0 @^b e_1 : t_2} \text{ if } b \leq \mathsf{top}(t_1) \ \wedge \ b \leq \mathsf{top}(t_2)$$

$$\frac{A \vdash e : t \rightarrow^b t}{A \vdash \mathtt{fix}^S \ e : t}$$

$$\frac{A \vdash e_0 : \mathtt{bool}^b \qquad A \vdash e_1 : t \qquad A \vdash e_2 : t}{A \vdash \mathtt{if}^b \ e_0 \ e_1 \ e_2 : t} \text{ if } b \leq \mathsf{top}(t)$$

$$\frac{A \vdash e : \mathtt{int}^S}{A \vdash e : \mathtt{int}^D} \qquad \frac{A \vdash e : \mathtt{bool}^S}{A \vdash e : \mathtt{bool}^D}$$

A.3 Multi-level partial evaluation: [1]

In this subsection we present a restriction of the binding time analysis of [1] (expressed using Scheme) to the lambda calculus of the present paper.

For types [1] defines a predicate $\vdash t : b$:

$$\frac{}{\vdash \mathtt{int}^b : b} \text{ if } 0 \leq b \leq \mathsf{max} \qquad \frac{}{\vdash \mathtt{bool}^b : b} \text{ if } 0 \leq b \leq \mathsf{max}$$

$$\frac{\vdash t_1 : b_1 \qquad \vdash t_2 : b_2}{\vdash t_1 \rightarrow^b t_2 : b} \text{ if } b_1 \ \geq \ b \wedge b_2 \ \geq \ b$$

Based on this define $\parallel t \parallel = b$ if and only if $\vdash t : b$.

For expressions the typing rules are:

$$\frac{}{A \vdash c_i^b : t} \text{ if } t = \mathsf{Type}(c_i^b) \qquad \frac{}{A \vdash x_i : t} \text{ if } t = A(x_i)$$

$$\frac{A[x_i : t_i] \vdash e : t}{A \vdash \lambda^b x_i.e : t_i \rightarrow^b t} \text{ if } \parallel t_i \parallel \geq \ b \qquad \frac{A \vdash e_0 : t_1 \rightarrow^b t_2 \qquad A \vdash e_1 : t_1}{A \vdash e_0 @^b e_1 : t_2}$$

$$\frac{A \vdash e : t \rightarrow^b t}{A \vdash \mathtt{fix}^0 \ e : t} \qquad \frac{A \vdash e_0 : \mathtt{bool}^b \qquad A \vdash e_1 : t \qquad A \vdash e_2 : t}{A \vdash \mathtt{if}^b \ e_0 \ e_1 \ e_2 : t} \text{ if } \parallel t \parallel \geq \ b$$

$$\frac{A \vdash e : \mathtt{int}^b}{A \vdash \mathtt{lift}_b^{b'} \ e : \mathtt{int}^{b+b'}} \text{ if } b + b' \leq \mathsf{max} \qquad \frac{A \vdash e : \mathtt{bool}^b}{A \vdash \mathtt{lift}_b^{b'} \ e : \mathtt{bool}^{b+b'}} \text{ if } b + b' \leq \mathsf{max}$$

Compared with [1] we have added a side condition to the rules for abstraction and lifting so as to ensure that the types derivable for the expressions are well-formed; this relates to the remark at the end of Subsection 3.2.

This article was processed using the LaTeX macro package with LLNCS style