Multi-Level Languages: a Descriptive Framework

Flemming Nielson and Hanne Riis Nielson

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 so as to conceal those differences between the systems that are not essential for the multi-level ideas, and so as to reveal the deeper similarities and differences. The multi-level λ-calculi defined here allow multi-level structures that are not restricted to the (possibly finite) linear orders found in most of the literature. Finally, we generalise our approach so as to be applicable to a much wider class of programming languages.

1 Introduction

Two-level languages are at least a decade old [9, 6] and multi-level languages at least four years old [13]. In particular two-level languages have been used extensively in the development of partial evaluation [2, 4] and abstract interpretation [7, 10] but also in areas such as code generation [11] and processor placement [14].

A main 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.

After presenting a few key definitions from algebra we proceed to define a notion of multi-level λ-calculi. This allows a very general structure upon the levels that is not restricted to be a (possibly finite) linearly ordered structure as is the case in most of the literature. We then consider five systems in detail and formally show that they can be formulated in our present framework. These systems are: a system for code generation [13], a system for

partial evaluation [4], a system for multi-level partial evaluation [2], a system for abstract interpretation [7], and a system based on modal logic [1]. An overview of parts of this development was previously reported in [15].

Finally, we pave the way for a much more general theory of multi-level languages by presenting definitions that are applicable for programming languages that are not necessarily based on the λ-calculus.

Remark about the choice of levels. 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 [13, 2]). 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 [13]) although a Kripke-structure [8] (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 [18, 3].

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 $(s_1 \cdots s_n; s)$. (Interpretations of Σ-algebras in other categories than **Set** can be found in the literature.)

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

[1] These approaches are not too dissimilar in their descriptive power and thus to some extent 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).

using the operators of Σ; as operators it has the constructions of 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 [3].)

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 [18] whenever it happens to be a total mapping.)

3 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 $\vdash^{\mathsf{T}}$:

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

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

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

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

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

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 about the choice of λ-calculus. 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:

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

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

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

where it is natural to include as an explicit condition that the argument type is well-formed. Since actually 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 does not matter whether we base ourselves on $\boldsymbol{\lambda}$ or on $\boldsymbol{\lambda}'$.

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 a $(W^B \times \{\mathsf{Typ}, \mathsf{Exp}\})$-sorted signature $\Omega^B = \Omega^B_e \cup \Omega^B_i$. Here

- Ω^B_e contains those operators that must be explicitly given,

whereas $\Omega_i^B = \{\iota^b_{s_1\cdots s_n;s} \mid b \in B \wedge s_1,\cdots,s_n,s \in \{\mathsf{Typ},\mathsf{Exp}\}\}$ contains those operators $\iota^b_{s_1\cdots s_n;s}$ of rank $((b,s_1)\cdots(b,s_n);(b,s))$ that we shall regard as implicitly present. 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 information:

- a $\{\mathsf{Typ},\mathsf{Exp}\}$-sorted signature Σ^L (defining the syntax of L); and
- a set R^L of labelled inference rules for the well-formedness judgements, where for simplicity of notation we allow distinct rules to share the same label; 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) its label 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[2] have $b' = b$; and

(ii) the rule $\delta(\Delta)$ is a permissible[3] rule in $\boldsymbol{\lambda}$.

We should point out that since the set R^L of rules usually is finite and the set Ω^B_i of implicitly given operators is infinite, the set Ω^B_i contains many operators for which there is no need; however, this presents no complications for our development.

3.1 Example: code generation [13]

We shall now show that the restriction of the two-level λ-calculus of [13] to $\boldsymbol{\lambda}$ (summarised in Appendix A.1) is an instance of the present framework. To this end we define the multi-level language $L = L_{cg}$.

[2] If the present choice about $b' = b$ turns out to be too restrictive it can be weakened to $b' \in \{b, b_1, \cdots, b_n\}$.

[3] 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. If we were to adopt one of the more restrictive possibilities then the choice between $\boldsymbol{\lambda}$ and $\boldsymbol{\lambda}'$ would be of importance.

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:

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

Note that we have two copies of every operator of Σ (except identifiers that can be viewed as place-holders). Annotation with r corresponds to the underlining notation used in [13] and annotation 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 $b = r$):

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

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

$$[UP] \quad \frac{\vdash_r^{\mathsf{T}} t_1 \rightarrow^r t_2}{\vdash_c^{\mathsf{T}} t_1 \rightarrow^r t_2}$$

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 3.1 $\vdash_b^{\mathsf{T}} t$ 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 [13] we shall let the type environment A associate a level b and a type t with each identifier x_i:

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

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

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

where as before[4] 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:

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

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

where $\mathsf{gr}(A) = \{(x_i^b : t) \mid A(x_i^b) = t\}$ is the *graph* of A.

Example 3.2 Consider the apply function with the following annotations:

$$\lambda^c f.\lambda^r x.f \mathtt{@}^r \, x$$

It has type

$$(\mathtt{int}^r \rightarrow^r \mathtt{int}^r) \rightarrow^c (\mathtt{int}^r \rightarrow^r \mathtt{int}^r)$$

The following inference tree shows that this indeed is a well-formed type at level c:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash^{\mathsf{T}}_r \mathtt{int}^r}[\iota^r] \quad \dfrac{}{\vdash^{\mathsf{T}}_r \mathtt{int}^r}[\iota^r]}{\vdash^{\mathsf{T}}_r \mathtt{int}^r \rightarrow^r \mathtt{int}^r}[\iota^r]}{\vdash^{\mathsf{T}}_c \mathtt{int}^r \rightarrow^r \mathtt{int}^r}[UP] \quad \dfrac{\dfrac{\dfrac{}{\vdash^{\mathsf{T}}_r \mathtt{int}^r}[\iota^r] \quad \dfrac{}{\vdash^{\mathsf{T}}_r \mathtt{int}^r}[\iota^r]}{\vdash^{\mathsf{T}}_r \mathtt{int}^r \rightarrow^r \mathtt{int}^r}[\iota^r]}{\vdash^{\mathsf{T}}_c \mathtt{int}^r \rightarrow^r \mathtt{int}^r}[UP]}{\vdash^{\mathsf{T}}_c (\mathtt{int}^r \rightarrow^r \mathtt{int}^r) \rightarrow^c (\mathtt{int}^r \rightarrow^r \mathtt{int}^r)}[\iota^c]$$

Note that the rule $[UP]$ is used to switch context: the upper parts of the inference tree are at level r and the lower parts at level c.

The inference tree below shows that the apply function has this type. Here A_f abbreviates $[f^c \mapsto \mathtt{int}^r \rightarrow^r \mathtt{int}^r]$ and A_{fx} abbreviates $[f^c \mapsto \mathtt{int}^r \rightarrow^r \mathtt{int}^r, x^r \mapsto \mathtt{int}^r]$. Again note how the rules $[up]$ and $[dn]$ are used to switch between the two levels.

[4] 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 [13] we shall assume that the entry *is* removed; however, it would be feasible to take the other approach (and then perhaps replace the operators $x_i \in \Sigma$ with $x_i^b \in \Sigma$) or perhaps to insist that all bound identifiers are distinct so that the choice does not matter.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_{fx}\vdash^{\mathsf{E}}_{c} f : \mathtt{int}^r\!\to^r\!\mathtt{int}^r}\,[\iota^c]}{A_{fx}\vdash^{\mathsf{E}}_{r} f : \mathtt{int}^r\!\to^r\!\mathtt{int}^r}\,[dn] \qquad \dfrac{}{A_{fx}\vdash^{\mathsf{E}}_{r} x : \mathtt{int}^r}\,[\iota^r]}{A_{fx}\vdash^{\mathsf{E}}_{r} f@^r\, x : \mathtt{int}^r}\,[\iota^r]}{A_f\vdash^{\mathsf{E}}_{r} \lambda^r x.f@^r\, x : \mathtt{int}^r\!\to^r\!\mathtt{int}^r}\,[\iota^r]}{A_f\vdash^{\mathsf{E}}_{c} \lambda^r x.f@^r\, x : \mathtt{int}^r\!\to^r\!\mathtt{int}^r}\,[up]}{[\,]\vdash^{\mathsf{E}}_{c} \lambda^c f.\lambda^r x.f@^r\, x : (\mathtt{int}^r\!\to^r\!\mathtt{int}^r)\!\to^c\!(\mathtt{int}^r\!\to^r\!\mathtt{int}^r)}\,[\iota^c]$$

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

Fact 3.3 $A\vdash^{\mathsf{E}}_{b} e : t$ implies $\vdash^{\mathsf{T}}_{b} t$.

To show that we have given the correct treatment for expressions we define a mapping $\langle\cdots\rangle$ into the type environments of Appendix A.1:

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

and we then prove:

Lemma 3.4 $A\vdash^{\mathsf{E}}_{b} e : t$ if and only if $\langle A\rangle \vdash e : t : b$ (in Appendix A.1).

Proof First we observe that it is straightforward to show that whenever $\langle A\rangle \vdash e : t : b$ then also $A\vdash^{\mathsf{E}}_{b} e : t$. For the other implication we prove the slightly stronger statement

$$A_1\vdash^{\mathsf{E}}_{b} e : t \text{ and } \mathsf{gr}(A_1)\subseteq\mathsf{gr}(A_2) \text{ imply } \langle A_2\rangle \vdash e : t : b.$$

The proof is by induction on the inference of $A_1\vdash^{\mathsf{E}}_{b} e : t$.

The cases of constants and identifiers are trivial using Fact 3.1.

For abstraction we have $A_1\vdash^{\mathsf{E}}_{b}\lambda^b x_i.e : t_i\!\to^b\!t$ because $A_1[x^b_i : t_i]\vdash^{\mathsf{E}}_{b} e : t$ and $\vdash^{\mathsf{T}}_{b} t_i$. The induction hypothesis gives $\langle A_2\rangle[x_i : (t_i : b)]\vdash e : t : b$ since $\mathsf{gr}(A_1[x^b_i : t_i])\subseteq\mathsf{gr}(A_2[x^b_i : t_i])$. Since Fact 3.1 gives $\vdash t_i : b$ we get $\langle A_2\rangle\vdash\lambda^b x_i.e : t_i\!\to^b\!t : b$ as required.

The cases of application, fixed points and conditional follow straightforwardly from the induction hypothesis.

In the case of $[dn]$ we have $A_1\vdash^{\mathsf{E}}_{r} e : t$ because $A'_1\vdash^{\mathsf{E}}_{c} e : t$, $\vdash^{\mathsf{T}}_{r} t$ and $\mathsf{gr}(A'_1)\subseteq\mathsf{gr}(A_1)$. Using the assumption $\mathsf{gr}(A_1)\subseteq\mathsf{gr}(A_2)$ we get $\mathsf{gr}(A'_1)\subseteq\mathsf{gr}(A_2)$ and the induction hypothesis gives $\langle A_2\rangle\vdash e : t : c$. From Fact 3.1 we get $\vdash t : r$ so $\langle A_2\rangle\vdash e : t : r$ as required.

In the case of $[up]$ we have $A_1\vdash^{\mathsf{E}}_{c} e : t$ because $A'_1\vdash^{\mathsf{E}}_{r} e : t$, $\vdash^{\mathsf{T}}_{c} t$, $\mathsf{gr}(A'_1)\subseteq\mathsf{gr}(A_1)$ and $\forall(x^{b'}_i : t')\in\mathsf{gr}(A'_1) : (b' = c\wedge\vdash^{\mathsf{T}}_{c} t')$. Now define A'_2 by $\mathsf{gr}(A'_2) = \{(x^{b'}_i, t')\in\mathsf{gr}(A_2)\mid b' = c\wedge\vdash^{\mathsf{T}}_{c} t'\}$. Since $\mathsf{gr}(A_1)\subseteq\mathsf{gr}(A_2)$ by assumption it follows that $\mathsf{gr}(A'_1)\subseteq\mathsf{gr}(A'_2)$. Thus the induction hypothesis gives $\langle A'_2\rangle\vdash e : t : r$. From Fact 3.1 we get $\vdash t : c$ so $\langle A_2\rangle\vdash e : t : c$ as required. □

To show that we have defined a multi-level λ-calculus we define a *uniform derivor* δ from L_{cg} into $\boldsymbol{\lambda}$: it simply removes all annotations. It is then fairly straightforward to prove:

Fact 3.5 L_{cg} is a multi-level λ-calculus.

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 that it is of no importance whether the derivor maps back to $\boldsymbol{\lambda}$ or $\boldsymbol{\lambda}'$.

Remark about the design decisions of [13]. As we shall see below, the multi-level λ-calculi developed for partial evaluation have bridging rules for types that are more permissive than the rule $[UP]$ of L_{cg}. It may therefore be appropriate to briefly recall the motivations behind the design of L_{cg} [9, 11, 13]. The idea is that a compiler manipulates code which when executed manipulates run-time values (like closures and lists); the compiler does not itself directly manipulate run-time values. This motivates the requirement that the compile-time level only involves the run-time functions rather than more general run-time data; to achieve the effect of operating on run-time data (say of type $\mathtt{int}^r$) one can instead operate on code that produces run-time data (say of type $\mathtt{unit}^r \to^r \mathtt{int}^r$). Technically, the correctness proofs of [11] depend profoundly on this property of L_{cg} and would seem not to apply to two-level languages without this property.

3.2 Example: partial evaluation [4]

We shall now show that the restriction of the binding time analysis of [4] to $\boldsymbol{\lambda}$ (summarised in Appendix A.2) is an instance of the present framework. To this end we define the multi-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[5]. 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.

[5]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. By the duality principle of lattice theory these differenes are only cosmetic.

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 (adhering to the design decisions of [4]) there is no $\mathtt{fix}^D$, i.e. all fix point computations must be static.

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

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

(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*:

$$[DN] \quad \frac{\vdash_D^{\mathsf{T}} t}{\vdash_S^{\mathsf{T}} t}$$

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 3.6 $\vdash_b^{\mathsf{T}} t$ if and only if $b \leq \mathsf{top}(t)$ (in Appendix A.2).

Proof First we prove that $\vdash_b^{\mathsf{T}} t$ implies $b \leq \mathsf{top}(t)$ by induction on the inference of $\vdash_b^{\mathsf{T}} t$. The cases of base types and function types are trivial since e.g. $\vdash_b^{\mathsf{T}} t_1 \rightarrow^b t_2$ implies that $\mathsf{top}(t_1 \rightarrow^b t_2) = b$ and we have $b \leq b$. The only non-trivial case is when $\vdash_S^{\mathsf{T}} t$ because $\vdash_D^{\mathsf{T}} t$. The induction hypothesis gives $D \leq \mathsf{top}(t)$ but since $S \leq D$ it follows that $S \leq \mathsf{top}(t)$ as required.

Next we prove that $b \leq \mathsf{top}(t)$ implies $\vdash_b^{\mathsf{T}} t$ by a case analysis on t. If t is a function type as e.g. $t_1 \rightarrow^{b_0} t_2$ then the assumption $b \leq \mathsf{top}(t_1 \rightarrow^{b_0} t_2)$ amounts to $b \leq b_0$. We trivially have $\vdash_{b_0}^{\mathsf{T}} t_1 \rightarrow^{b_0} t_2$ and thus have the result unless $b \neq b_0$. But then $b_0 = D$ and $b = S$ and the result follows from the rule $[DN]$. The cases of base types are similar. □

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

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

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

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

(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; furthermore, note that the application rule now has a side condition ensuring that the result type is well-formed. In addition we have the following bridging rules corresponding to the operators *up*, *dn* and *coer*:

$$[up]\quad \frac{A\vdash^{\mathsf{E}}_{S} e : t}{A\vdash^{\mathsf{E}}_{D} e : t} \text{ if } \vdash^{\mathsf{T}}_{D} t \qquad [dn]\quad \frac{A\vdash^{\mathsf{E}}_{D} e : t}{A\vdash^{\mathsf{E}}_{S} e : t}$$

$$[coer]\quad \frac{A\vdash^{\mathsf{E}}_{S} e : \mathtt{int}^{S}}{A\vdash^{\mathsf{E}}_{D} e : \mathtt{int}^{D}} \qquad [coer]\quad \frac{A\vdash^{\mathsf{E}}_{S} e : \mathtt{bool}^{S}}{A\vdash^{\mathsf{E}}_{D} e : \mathtt{bool}^{D}}$$

Note that the rule [*coer*] has no counterpart in Subsection 3.1.

Example 3.7 We shall now illustrate how a static conditional with two dynamic branches can be typed dynamically. Consider the expression:

$$\mathtt{if}^{S}\ e_0\ e_1\ e_2$$

where e.g. $A\vdash^{\mathsf{E}}_{S} e_0 : \mathtt{bool}^{S}$, $A\vdash^{\mathsf{E}}_{D} e_1 : \mathtt{int}^{D}\!\rightarrow^{D}\mathtt{int}^{D}$ and $A\vdash^{\mathsf{E}}_{D} e_2 : \mathtt{int}^{D}\!\rightarrow^{D}\mathtt{int}^{D}$. We have the following inference tree:

$$\dfrac{\dfrac{\vdots}{A\vdash^{\mathsf{E}}_{S} e_0 : \mathtt{bool}^{S}} \quad \dfrac{\dfrac{\vdots}{A\vdash^{\mathsf{E}}_{D} e_1 : \mathtt{int}^{D}\!\rightarrow^{D}\mathtt{int}^{D}}}{A\vdash^{\mathsf{E}}_{S} e_1 : \mathtt{int}^{D}\!\rightarrow^{D}\mathtt{int}^{D}}\,[dn] \quad \dfrac{\dfrac{\vdots}{A\vdash^{\mathsf{E}}_{D} e_2 : \mathtt{int}^{D}\!\rightarrow^{D}\mathtt{int}^{D}}}{A\vdash^{\mathsf{E}}_{S} e_2 : \mathtt{int}^{D}\!\rightarrow^{D}\mathtt{int}^{D}}\,[dn]}{A\vdash^{\mathsf{E}}_{S} \mathtt{if}^{S}\ e_0\ e_1\ e_2 : \mathtt{int}^{D}\!\rightarrow^{D}\mathtt{int}^{D}}\,[\iota^{S}]}{A\vdash^{\mathsf{E}}_{D} \mathtt{if}^{S}\ e_0\ e_1\ e_2 : \mathtt{int}^{D}\!\rightarrow^{D}\mathtt{int}^{D}}\,[up]$$

Note that in order to apply the rule for the conditional the judgements of the two branches are transfered to the static level using [*dn*] and later the overall judgement of the conditional is transfered back to the dynamic level using [*up*].

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

Fact 3.8 $A\vdash^{\mathsf{E}}_{b} e : t$ implies $\vdash^{\mathsf{T}}_{b} t$.

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

Lemma 3.9 $A\vdash^{\mathsf{E}}_{b} e : t$ if and only if $A \vdash e : t \ \wedge\ b \leq \mathsf{top}(t)$ (in Appendix A.2).

Proof First we prove that if $A\vdash^{\mathsf{E}}_{b} e : t$ then $A \vdash e : t$ and $b \leq \mathsf{top}(t)$. We proceed by induction on the inference of $A\vdash^{\mathsf{E}}_{b} e : t$.

The cases of constants and identifies follow trivially using Fact 3.6.

In the case of abstraction we have $A\vdash^{\mathsf{E}}_{b} \lambda^{b} x_i.e : t_i\!\rightarrow^{b}\!t$ because $A[x_i : t_i]\vdash^{\mathsf{E}}_{b} e : t$ and $\vdash^{\mathsf{T}}_{b} t_i$. The induction hypothesis gives $A[x_i : t_i] \vdash e : t$ and $b \leq \mathsf{top}(t)$. Fact 3.6 gives $b \leq \mathsf{top}(t_i)$ so we get $A \vdash \lambda^{b} x_i.e : t_i\!\rightarrow^{b}\!t$. Clearly $b \leq \mathsf{top}(t_i\!\rightarrow^{b}\!t)$.

In the case of application we have $A\vdash^{\mathsf{E}}_{b} e_0 @^{b} e_1 : t_2$ because $A\vdash^{\mathsf{E}}_{b} e_0 : t_1\!\rightarrow^{b}\!t_2$, $A\vdash^{\mathsf{E}}_{b} e_1 : t_1$ and $\vdash^{\mathsf{T}}_{b} t_2$. The induction hypothesis gives $A \vdash e_0 : t_1\!\rightarrow^{b}\!t_2$, $A \vdash e_1 : t_1$ and $b \leq \mathsf{top}(t_1)$. Fact 3.6 gives $b \leq \mathsf{top}(t_2)$. Thus we get $A \vdash e_0 @^{b} e_1 : t_2$ as required.

The cases of fixed points and conditional follow straightforwardly from the induction hypothesis. Similarly for the rules [*up*] and [*dn*].

In the case of a [*coer*] rule we e.g. have $A \vdash^{\mathsf{E}}_{b} e : \mathtt{int}^{D}$ because $A \vdash^{\mathsf{E}}_{b} e : \mathtt{int}^{S}$. The induction hypothesis gives $A \vdash e : \mathtt{int}^{S}$ and $b \leq S$. Clearly $A \vdash e : \mathtt{int}^{D}$ and since $S \leq D$ we get $b \leq D$ as required.

Next we prove that if $A \vdash e : t$ and $b_0 \leq \mathsf{top}(t)$ then $A \vdash^{\mathsf{E}}_{b_0} e : t$. We proceed by induction on the inference of $A \vdash e : t$.

The cases of constants and identifiers are trivial since Fact 3.6 applied to the assumption $b_0 \leq \mathsf{top}(t)$ gives $\vdash^{\mathsf{T}}_{b_0} t$.

For abstraction we assume that $A \vdash \lambda^{b} x.e : t_i \rightarrow^{b} t$ because $A[x_i : t_i] \vdash e : t$, $b \leq \mathsf{top}(t_i)$ and $b \leq \mathsf{top}(t)$ and furthermore we assume that $b_0 \leq \mathsf{top}(t_i \rightarrow^{b} t)$ (i.e. $b_0 \leq b$). The induction hypothesis gives $A[x_i : t_i] \vdash^{\mathsf{E}}_{b} e : t$. From Fact 3.6 we get $\vdash^{\mathsf{T}}_{b} t_i$ so we have $A \vdash^{\mathsf{E}}_{b} \lambda^{b} x.e : t_i \rightarrow^{b} t$. If $b = b_0$ we are finished and otherwise $b_0 = S$ and $b = D$. Clearly then the rule [*dn*] can be applied and gives the required result.

The cases of application, fixed points and conditional follow from the induction hypothesis in a similar way.

Finally consider the coercion rules: Assume $A \vdash e : \mathtt{int}^{D}$ because $A \vdash e : \mathtt{int}^{S}$ and furthermore assume that $b_0 \leq \mathsf{top}(\mathtt{int}^{D})$ (i.e. $b_0 \leq D$). The induction hypothesis gives $A \vdash^{\mathsf{E}}_{S} e : \mathtt{int}^{S}$ and thereby $A \vdash^{\mathsf{E}}_{D} e : \mathtt{int}^{D}$ using the rule [*coer*]. If $b_0 = D$ then we are finished, otherwise $b_0 = S$ and the rule [*dn*] will give the required result. □

To show that we have defined a multi-level λ-calculus we define a uniform derivor δ from L_{pe} into $\boldsymbol{\lambda}$: it simply removes all annotations. It is then fairly straightforward to prove:

Fact 3.10 L_{pe} is a multi-level λ-calculus.

Remark about the design decisions of [4]. 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 [4] one can in fact prove

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

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

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

thus bringing the system closer to that of Subsection 3.1. As a consequence we can remove the side condition $\vdash^{\mathsf{T}}_{b} t_2$ 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} and would expect it to be more useful than L_{pe}.

3.3 Example: multi-level partial evaluation [2]

We shall now show that the restriction of the multi-level binding time analysis of [2] to $\boldsymbol{\lambda}$ (summarised in Appendix A.3) is an instance of the present framework. To this end we define the multi-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 $0 \leq b < b+b' \leq \mathsf{max}$
- $dn_b^{b'}$: $((b+b', \mathsf{Exp}); (b, \mathsf{Exp}))$ for $0 \leq b < b+b' \leq \mathsf{max}$
- $up_b^{b'}$, $lift_b^{b'}$: $((b, \mathsf{Exp}); (b+b', \mathsf{Exp}))$ for $0 \leq b < b+b' \leq \mathsf{max}$

Thus $DN_b^{b'}$ allows us to embed types at level $b+b'$ at the lower level b; this imposes the ordering that $b < b+b'$ 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 $b \in \{0, 1, \cdots, \mathsf{max}\}$:

$\rightarrow^b : (\mathsf{Typ}^2; \mathsf{Typ})$	$\mathtt{int}^b : (; \mathsf{Typ})$	$\mathtt{bool}^b : (; \mathsf{Typ})$
$c_i^b : (; \mathsf{Exp})$	$x_i : (; \mathsf{Exp})$	$\lambda^b x_i. : (\mathsf{Exp}; \mathsf{Exp})$
$@^b : (\mathsf{Exp}^2; \mathsf{Exp})$	$\mathtt{if}^b : (\mathsf{Exp}^3; \mathsf{Exp})$	$\mathtt{fix}^0 : (\mathsf{Exp}; \mathsf{Exp})$
$\mathtt{lift}_b^{b'} : (\mathsf{Exp}; \mathsf{Exp})$	for $0 \leq b < b+b' \leq \mathsf{max}$	

As in Subsection 3.2 (adhering to the design decisions of [2]) all fix point computations are required to be static[6], i.e. at level 0. Note that in addition to the annotations on the operators of $\boldsymbol{\lambda}$ we also have the new operators $\mathtt{lift}_b^{b'}$ which are explicit coercion operators.

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

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

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

$$[DN_b^{b'}] \quad \frac{\vdash_{b+b'}^{\mathsf{T}} t}{\vdash_b^{\mathsf{T}} t}$$

[6] In [2] recursive computations are specified implicitly.

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 3.11 $\vdash_b^{\mathsf{T}} t$ if and only if $\| t \| \geq b$ (in Appendix A.3).

Proof First we prove that $\vdash_b^{\mathsf{T}} t$ implies $\| t \| \geq b$ by induction on the inference of $\vdash_b^{\mathsf{T}} t$. The case of base types is straightforward. Next assume that $\vdash_b^{\mathsf{T}} t_1 \rightarrow^b t_2$ because $\vdash_b^{\mathsf{T}} t_1$ and $\vdash_b^{\mathsf{T}} t_2$. The induction hypothesis gives $\| t_1 \| \geq b$ and $\| t_2 \| \geq b$ meaning that $\vdash t_1 : b_1$ and $\vdash t_2 : b_2$ for $b_1 \geq b$ and $b_2 \geq b$. Thus $\vdash t_1 \rightarrow^b t_2 : b$ and $\| t_1 \rightarrow^b t_2 \| = b$. The case where $\vdash_b^{\mathsf{T}} t$ because $\vdash_{b+b'}^{\mathsf{T}} t$ follows directly from the induction hypothesis.

To prove the other implication assume that $\| t \| \geq b$, i.e. that $\vdash t : b'$ and $b' \geq b$. By induction on the inference of $\vdash t : b'$ we shall then prove that $\vdash_b^{\mathsf{T}} t$. The case where t is a base type is straightforward using rule $[DN_b^{b'-b}]$ whenever $b' > b$. In the inductive case we assume that $\vdash t_1 \rightarrow^b t_2 : b$ because $\vdash t_1 : b_1$, $\vdash t_2 : b_2$, $b_1 \geq b$ and $b_2 \geq b$. The induction hypothesis gives $\vdash_b^{\mathsf{T}} t_1$ and $\vdash_b^{\mathsf{T}} t_2$ and it follows that $\vdash_b^{\mathsf{T}} t_1 \rightarrow^b t_2$. □

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

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

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

$$[\iota^b] \quad \frac{A \vdash_0^{\mathsf{E}} e : t \rightarrow^b t}{A \vdash_0^{\mathsf{E}} \mathtt{fix}^0\ e : t} \qquad [\iota^b] \quad \frac{A \vdash_b^{\mathsf{E}} e_0 : \mathtt{bool}^b \quad A \vdash_b^{\mathsf{E}} e_1 : t \quad A \vdash_b^{\mathsf{E}} e_2 : t}{A \vdash_b^{\mathsf{E}} \mathtt{if}^b\ e_0\ e_1\ e_2 : t}$$

(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 $lift_b^{b'}$:

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

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

If we restrict ourselves to just two levels then we have the multi-level language L'_{pe} of Subsection 3.2 except that L'_{pe} has no syntactic operators to express coercion.

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

Fact 3.12 $A \vdash_b^{\mathsf{E}} e : t$ implies $\vdash_b^{\mathsf{T}} t$.

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

Lemma 3.13 $A \vdash_b^{\mathsf{E}} e : t$ if and only if $A \vdash e : t \wedge \| t \| \geq b$ (in Appendix A.3).

Proof First we prove that if $A \vdash_b^{\mathsf{E}} e : t$ then $A \vdash e : t$ and $\| t \| \geq b$. We proceed by induction on the inference of $A \vdash_b^{\mathsf{E}} e : t$.

The cases of constants and identifiers follow trivially from Fact 3.11.

In the case of abstraction we have $A \vdash_b^{\mathsf{E}} \lambda^b x_i.e : t_i \rightarrow^b t$ because $A[x_i : t_i] \vdash_b^{\mathsf{E}} e : t$ and $\vdash_b^{\mathsf{T}} t_i$. The induction hypothesis gives $A[x_i : t_i] \vdash e : t$ and $\| t \| \geq b$. From Fact 3.11 we get $\| t_i \| \geq b$ so $A \vdash \lambda^b x_i.e : t_i \rightarrow^b t$. Clearly $\| t_1 \rightarrow^b t_2 \| \geq b$.

The cases of application, fixed points and conditional follow straightforwardly from the induction hypothesis. Similarly for $[up]$ and $[dn]$.

In the case of a $[lift]$ rule we e.g. have $A \vdash^{\mathsf{E}}_{b+b'} \mathtt{lift}^{b'}_{b}\ e : \mathtt{int}^{b+b'}$ because $A \vdash^{\mathsf{E}}_{b} e : \mathtt{int}^{b}$ and $b < b + b' \leq \mathsf{max}$. The induction hypothesis gives $A \vdash e : \mathtt{int}^{b}$ and $\parallel \mathtt{int}^{b} \parallel \geq b$. Clearly $A \vdash \mathtt{lift}^{b'}_{b}\ e : \mathtt{int}^{b+b'}$ and $\parallel \mathtt{int}^{b+b'} \parallel \geq b + b'$ as required.

Next we prove that if $A \vdash e : t$ and $\parallel t \parallel \geq b_0$ then $A \vdash^{\mathsf{E}}_{b_0} e : t$. We proceed by induction on the inference of $A \vdash e : t$.

The cases of constants and identifiers are trivial since Fact 3.11 applied to the assumption $\parallel t \parallel \geq b_0$ gives $\vdash^{\mathsf{T}}_{b_0} t$.

For abstraction we assume that $A \vdash \lambda^{b} x.e : t_i \rightarrow^{b} t$ because $A[x_i : t_i] \vdash e : t$ and $\parallel t_i \parallel \geq b$ and furthermore we assume that $\parallel t_i \rightarrow^{b} t \parallel \geq b_0$. Then $\parallel t_i \parallel \geq b$ ($\geq b_0$) and $\parallel t \parallel \geq b$ ($\geq b_0$). The induction hypothesis gives $A[x_i : t_i] \vdash^{\mathsf{E}}_{b} e : t$ and Fact 3.11 gives $\vdash^{\mathsf{T}}_{b} t_i$ and hence $A \vdash^{\mathsf{E}}_{b} \lambda^{b} x.e : t_i \rightarrow^{b} t$. Since Fact 3.11 gives $\vdash^{\mathsf{T}}_{b_0} t_i \rightarrow^{b} t$ we get $A \vdash^{\mathsf{E}}_{b_0} \lambda^{b} x.e : t_i \rightarrow^{b} t$ using the rule $[dn^{b-b_0}_{b_0}]$ whenever $b > b_0$.

The cases of application, fixed points and conditional follows from the induction hypothesis in a similar way.

Finally consider the coercion rules: Assume $A \vdash \mathtt{lift}^{b'}_{b}\ e : \mathtt{int}^{b+b'}$ because $A \vdash e : \mathtt{int}^{b}$ and $b < b + b' \leq \mathsf{max}$ and furthermore that $\parallel \mathtt{int}^{b+b'} \parallel \geq b_0$. The induction hypothesis gives $A \vdash^{\mathsf{E}}_{b} e : \mathtt{int}^{b}$ and thereby $A \vdash^{\mathsf{E}}_{b+b'} \mathtt{lift}^{b'}_{b}\ e : \mathtt{int}^{b+b'}$. From Fact 3.11 we get $\vdash^{\mathsf{T}}_{b_0} \mathtt{int}^{b+b'}$ so we get that $A \vdash^{\mathsf{E}}_{b_0} \mathtt{lift}^{b'}_{b}\ e : \mathtt{int}^{b+b'}$ using the rule $[dn^{b+b'-b_0}_{b_0}]$ whenever $b + b' > b_0$. □

To show that we have defined a multi-level λ-calculus we define a uniform derivor δ from L_{mp} into $\boldsymbol{\lambda}$: it simply removes all annotations and all occurrences of $\mathtt{lift}^{b'}_{b}$. It is then fairly straightforward to prove:

Fact 3.14 L_{mp} is a multi-level λ-calculus.

3.4 Example: abstract interpretation [7]

We shall now show that the two-level language TML[dt,dt] of [7] 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 [7] where the combinators are replaced by λ-expressions; consequently it will be instructive to think only of forward program analyses and we shall dispense with proving a formal relationship between L_{ai} and TML[dt,dt]. Given these considerations we can define the multi-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:

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

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:

$$[\iota^b] \quad \frac{}{\vdash_b^{\mathsf{T}} \mathtt{int}^b} \qquad [\iota^b] \quad \frac{}{\vdash_b^{\mathsf{T}} \mathtt{bool}^b} \qquad [\iota^b] \quad \frac{\vdash_b^{\mathsf{T}} t_1 \quad \vdash_b^{\mathsf{T}} t_2}{\vdash_b^{\mathsf{T}} t_1 \to^b t_2}$$

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

$$[UP] \quad \frac{\vdash_l^{\mathsf{T}} t}{\vdash_d^{\mathsf{T}} t}$$

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

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

that has no counterpart in Subsections 3.1 and 3.2; it reflects that a function space from a domain to a lattice is indeed a lattice. It is straightforward to show that $\vdash_d^{\mathsf{T}} t'$ holds if and only if $\mathbf{dt}(t')$ holds in [7], and that $\vdash_l^{\mathsf{T}} t'$ holds if and only if $\mathbf{lt}(t')$ holds in [7].

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

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

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

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

(where b ranges over $\{l, d\}$); these rules are exactly as in Subsection 3.1. On top of this we have the two bridging rules

$$[dn] \quad \frac{A \vdash^{\mathsf{E}}_{d} e : t}{A \vdash^{\mathsf{E}}_{l} e : t} \text{ if } \vdash^{\mathsf{T}}_{l} t$$

$$[up] \quad \frac{A \vdash^{\mathsf{E}}_{l} e : t}{A \vdash^{\mathsf{E}}_{d} e : t}$$

which corresponds to two of the bridging rules in Subsection 3.2. It is trivial to establish the following relationship between the typing judgements:

Fact 3.15 $A \vdash^{\mathsf{E}}_{b} e : t$ implies $\vdash^{\mathsf{T}}_{b} t$.

To show that we have defined a two-level λ-calculus we define a uniform derivor δ: as in the previous examples it simply removes all annotations. It is then fairly straightforward to prove:

Fact 3.16 L_{ai} is a multi-level λ-calculus.

Comparison. It is instructive to consider the relationship between L_{ai} and the systems L_{cg} and L'_{pe}. Clearly we have the analogies

$$l \sim r \sim D \qquad (\bot)$$
$$d \sim c \sim S \qquad (\top)$$

and let us now consider the bridging rules between types. The L_{ai} system is unique in allowing to embed types of level $(\top)$ into types of level $(\bot)$ which (given the formulation of the rule $[DN]$ of L_{ai}) is perfectly safe for abstract interpretation, whereas it would be mind-boggling for code generation as well as partial evaluation. Apart from this the system L_{ai} has the general bridging rule also found in L'_{pe} for embedding types of level $(\bot)$ into types of level $(\top)$ and thus does not need the more restricted bridging rule found in L_{cg}. A similar comment holds for the bridging rules for expressions.

3.5 Example: modal language [1]

We shall now show that the restrictions of the modal logic languages $\mathsf{MiniML}^{\Box}_{K}$ and $\mathsf{MiniML}^{\Box}$ to $\boldsymbol{\lambda}$ (both summarised in Appendix A.4) are instances of the present framework. To this end we define the multi-level languages L_{mlK} and L_{ml}.

Multi-level structure. Let B be the set $\{0, 1, \cdots\}$ of levels where intuitively 0 corresponds to compile-time or static. The signature Ω^{B}_{mlK} then has the following explicitly given operators:

- $BOX^{b'}_{b}$: $((b, \mathsf{Typ}); (b', \mathsf{Typ}))$ for $0 \leq b' = b - 1$
- $box^{b'}_{b}$: $((b, \mathsf{Exp}); (b', \mathsf{Exp}))$ for $0 \leq b' = b - 1$
- $unbox1^{b'}_{b}$: $((b', \mathsf{Exp}); (b, \mathsf{Exp}))$ for $0 \leq b' = b - 1$

(so $b = b' + 1 \geq 1$). The signature Ω^B_{ml} does not contain the operator $unbox1$ but instead has:

- $pop^{b'}_b$: $((b, \mathsf{Exp}); (b', \mathsf{Exp}))$ for $0 \leq b' = b - 1$
- $unbox^b_b$: $((b, \mathsf{Exp}); (b, \mathsf{Exp}))$

Here we prefer to highlight the latter as an explicit operator rather than keeping it as an implicit one.

Multi-level λ-calculus. We use the following signature Σ^L_{mlK}:

$\rightarrow$: $(\mathsf{Typ}^2; \mathsf{Typ})$ $\quad$ $\mathtt{int} : (; \mathsf{Typ})$ $\quad$ $\mathtt{bool} : (; \mathsf{Typ})$

$\Box : (\mathsf{Typ}; \mathsf{Typ})$

$c_i : (; \mathsf{Exp})$ $\quad$ $x_i : (; \mathsf{Exp})$ $\quad$ $\lambda x_i. : (\mathsf{Exp}; \mathsf{Exp})$

$\mathtt{@} : (\mathsf{Exp}^2; \mathsf{Exp})$ $\quad$ $\mathtt{if} : (\mathsf{Exp}^3; \mathsf{Exp})$ $\quad$ $\mathtt{fix} : (\mathsf{Exp}; \mathsf{Exp})$

$\mathtt{box} : (\mathsf{Exp}; \mathsf{Exp})$ $\quad$ $\mathtt{unbox1} : (\mathsf{Exp}; \mathsf{Exp})$

The signature Σ^L_{ml} does not contain `unbox1` but instead has:

$\mathtt{pop} : (\mathsf{Exp}; \mathsf{Exp})$ $\quad$ $\mathtt{unbox} : (\mathsf{Exp}; \mathsf{Exp})$

Note that the operators are not annotated with levels as was the case in the previous Subsections.

For *types* we have the following well-formedness rules:

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

and the following bridging rule:

$$[BOX^b_{b+1}] \quad \frac{\vdash^{\mathsf{T}}_{b+1} t}{\vdash^{\mathsf{T}}_b \Box t}$$

One can then prove that we have given the correct treatment of types:

Fact 3.17 $\vdash^{\mathsf{T}}_b t$ if and only if $\vdash t$ (in Appendix A.4).

Proof We first prove that $\vdash^{\mathsf{T}}_b t$ if and only if $\vdash^{\mathsf{T}}_{b+1} t$ by induction on the inference tree; for this we use that $b \in B$ implies $(b+1) \in B$. We next show that $\vdash^{\mathsf{T}}_b t$ always holds by *reductio ad absurdum*: otherwise there must be a type t with fewest symbols such that $\vdash^{\mathsf{T}}_b t$ fails for some b, and clearly t cannot be of the form `int`, `bool`, $t_1 \rightarrow t_2$ or $\Box t_0$. $\Box$

For *expressions* we have the following well-formedness rules adapted from $\boldsymbol{\lambda}'$:

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

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

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

Note that the type environment associates types as well as levels with the identifiers although the expressions are not annotated. Also note that all occurrences of $\vdash^{\mathsf{T}}_b t$ are vacuously fulfilled and hence could be dispensed with. To introduce the bridging rules we need a little auxiliary notation:

$$\Vdash_b A \text{ if and only if } \forall (x_i^{b'} : t') \ \in \ \mathsf{gr}(A) : b' \leq b$$
$$\mathsf{gr}(A) \ominus b = \{(x_i^{b'} : t') \ \in \ \mathsf{gr}(A) \mid b \ \neq \ b'\}$$

Then L_{mlK} has the bridging rules:

$$[box^b_{b+1}] \quad \frac{A_1 \vdash^{\mathsf{E}}_{b+1} e : t}{A_2 \vdash^{\mathsf{E}}_b \texttt{box } e : \Box t} \text{ if } \mathsf{gr}(A_1) = \mathsf{gr}(A_2) \ \wedge \ \Vdash_b A_2$$

$$[unbox1^b_{b+1}] \quad \frac{A_1 \vdash^{\mathsf{E}}_b e : \Box t}{A_2 \vdash^{\mathsf{E}}_{b+1} \texttt{unbox1 } e : t} \text{ if } \mathsf{gr}(A_1) = (\mathsf{gr}(A_2) \ominus (b+1)) \ \wedge \ \Vdash_{b+1} A_2$$

whereas in the case of L_{ml} we do not have the rule $[unbox1]$ but instead have:

$$[pop^b_{b+1}] \quad \frac{A_1 \vdash^{\mathsf{E}}_b e : \Box t}{A_2 \vdash^{\mathsf{E}}_{b+1} \texttt{pop } e : \Box t} \text{ if } \mathsf{gr}(A_1) = (\mathsf{gr}(A_2) \ominus (b+1)) \ \wedge \ \Vdash_{b+1} A_2$$

$$[unbox^b_b] \quad \frac{A_1 \vdash^{\mathsf{E}}_b e : \Box t}{A_2 \vdash^{\mathsf{E}}_b \texttt{unbox } e : t} \text{ if } \mathsf{gr}(A_1) = \mathsf{gr}(A_2)$$

Since all types are well-formed it is trivial to establish the following relationship between the typing judgements:

Fact 3.18 $A \vdash^{\mathsf{E}}_b e : t$ implies $\vdash^{\mathsf{T}}_b t$.

To show that we have given the correct treatment for expressions we define a mapping[7] $\langle \cdots \rangle$ upon the type environments of Appendix A.4:

$$\mathsf{gr}(\langle \Gamma_0 \cdots \Gamma_b \rangle) = \{(x_i^{b'} : t') \mid b' \ \in \ \{0, 1, \cdots, b\} \ \wedge \ t' = \Gamma_{b'}(x_i)\}$$

and we then prove:

Lemma 3.19 $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash^{\mathsf{E}}_b e \ : \ t$ (in L_{mlK} resp. L_{ml}) if and only if $\Gamma_0 \cdots \Gamma_b \vdash \ e \ : \ t$ (in $\mathsf{MiniML}^{\Box}_K$ resp. $\mathsf{MiniML}^{\Box}$ of Appendix A.4).

[7] Actually there is a small sublety here concerning $\mathsf{gr}(A)$ in L_{mlK} and L_{ml}: is it permissible for $\mathsf{gr}(A)$ to contain both $(x_i^{b'} : t')$ and $(x_i^{b''} : t'')$ when $t' \ \neq \ t''$ and $b' \ \neq \ b''$? To obtain a succinct formulation of the relationship with [1] we allow this phenomenon; if we were to disallow it and stick to the decision of Subsection 3.1, we would instead require that all bound identifiers must be distinct.

Proof First we prove that if $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e : t$ then $\Gamma_0 \cdots \Gamma_b \vdash e : t$. We proceed by induction on the inference of $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e : t$ and deal with L_{mlK} and L_{ml} simultaneously.

The cases of constants and identifiers are immediate.

In the case of abstraction we have $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} \lambda x_i.e : t_i \to t$ because $\vdash_b^{\mathsf{T}} t_i$ and $\langle \Gamma_0 \cdots \Gamma_b \rangle [x_i^b : t_i] \vdash_b^{\mathsf{E}} e : t$. Since $\langle \Gamma_0 \cdots \Gamma_b \rangle [x_i^b : t_i]$ equals $\langle \Gamma_0 \cdots (\Gamma_b[x_i : t_i]) \rangle$ the induction hypothesis gives $\Gamma_0 \cdots (\Gamma_b[x_i : t_i]) \vdash e : t$ from which the desired $\Gamma_0 \cdots \Gamma_b \vdash \lambda x_i.e : t_i \to t$ follows.

In the case of application we have $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e_0 @ e_1 : t_2$ because $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e_0 : t_1 \to t_2$ and $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e_1 : t_1$. The induction hypotheses then give $\Gamma_0 \cdots \Gamma_b \vdash e_0 : t_1 \to t_2$ and $\Gamma_0 \cdots \Gamma_b \vdash e_1 : t_1$ from which the desired $\Gamma_0 \cdots \Gamma_b \vdash e_0 @ e_1 : t_2$ follows.

The cases of fixed points and conditional are straightforward.

In the case of `box` we have $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} \texttt{box}\ e : \Box t$ because $\Vdash_b \langle \Gamma_0 \cdots \Gamma_b \rangle$ and $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_{b+1}^{\mathsf{E}} e : t$. Setting $\Gamma_{b+1} = [\,]$ we clearly have $\langle \Gamma_0 \cdots \Gamma_b \rangle = \langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle$. The induction hypothesis then gives $\Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \vdash e : t$ from which the desired $\Gamma_0 \cdots \Gamma_b \vdash \texttt{box}\ e : \Box t$ follows.

In the case of `unbox1` we have $\langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle \vdash_{b+1}^{\mathsf{E}} \texttt{unbox1}\ e : t$ because $\Vdash_{b+1} \langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle$ and $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e : \Box t$ where we used that $\mathsf{gr}(\langle \Gamma_0 \cdots \Gamma_b \rangle) = (\mathsf{gr}(\langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle) \ominus (b+1))$. The induction hypothesis then gives $\Gamma_0 \cdots \Gamma_b \vdash e : \Box t$ and the desired $\Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \vdash \texttt{unbox1}\ e : t$ follows.

In the case of `pop` we have $\langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle \vdash_{b+1}^{\mathsf{E}} \texttt{pop}\ e : \Box t$ because $\Vdash_{b+1} \langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle$ and $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e : \Box t$. The induction hypothesis then gives $\Gamma_0 \cdots \Gamma_b \vdash e : \Box t$ and the desired $\Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \vdash \texttt{pop}\ e : \Box t$ follows.

In the case of `unbox` we have $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} \texttt{unbox}\ e : t$ because $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e : \Box t$. The induction hypothesis then gives $\Gamma_0 \cdots \Gamma_b \vdash e : \Box t$ and the desired result $\Gamma_0 \cdots \Gamma_b \vdash \texttt{unbox}\ e : t$ follows.

Next we prove that if $\Gamma_0 \cdots \Gamma_b \vdash e : t$ then $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e : t$. We proceed by induction on the inference of $\Gamma_0 \cdots \Gamma_b \vdash e : t$ and deal with $\mathsf{MiniML}_K^{\Box}$ and $\mathsf{MiniML}^{\Box}$ simultaneously.

The cases of constants and identifiers are immediate since $\vdash_b^{\mathsf{T}} t$ holds vacuously (by Fact 3.17).

In the case for abstraction we have $\Gamma_0 \cdots \Gamma_b \vdash \lambda x_i.e : t_i \to t$ because $\Gamma_0 \cdots (\Gamma_b[x_i : t_i]) \vdash e : t$. We have ensured that $\langle \Gamma_0 \cdots (\Gamma_b[x_i : t_i]) \rangle$ is defined and it clearly equals $\langle \Gamma_0 \cdots \Gamma_b \rangle [x_i^b : t_i]$. The induction hypothesis therefore gives $\langle \Gamma_0 \cdots \Gamma_b \rangle [x_i^b : t_i] \vdash_b^{\mathsf{E}} e : t$ and the desired result $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} \lambda x_i.e : t_i \to t$ follows because $\vdash_b^{\mathsf{T}} t_i$ holds vacuously (by Fact 3.17).

In the case for application we have $\Gamma_0 \cdots \Gamma_b \vdash e_0 @ e_1 : t_2$ because $\Gamma_0 \cdots \Gamma_b \vdash e_0 : t_1 \to t_2$ and $\Gamma_0 \cdots \Gamma_b \vdash e_1 : t_1$. The induction hypotheses then give $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e_0 : t_1 \to t_2$ and $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e_1 : t_1$ and the desired result $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} e_0 @ e_1 : t_2$ follows.

The cases of fixed points and conditional are straightforward.

In the case of `box` we have $\Gamma_0 \cdots \Gamma_b \vdash \texttt{box}\ e : \Box t$ because $\Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \vdash e : t$ where $\Gamma_{b+1} = [\,]$. The induction hypothesis then gives $\langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle \vdash_{b+1}^{\mathsf{E}} e : t$. Since $\langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle = \langle \Gamma_0 \cdots \Gamma_b \rangle$ and $\Vdash_b \langle \Gamma_0 \cdots \Gamma_b \rangle$ is immediate, the desired result $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash_b^{\mathsf{E}} \texttt{box}\ e : \Box t$ follows.

In the case of `unbox1` we have $\Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \vdash \texttt{unbox1}\ e : t$ because $\Gamma_0 \cdots \Gamma_b \vdash e : \Box t$. The induction hypothesis then gives $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash^{\mathsf{E}}_b e : \Box t$. Since $\mathsf{gr}(\langle \Gamma_0 \cdots \Gamma_b \rangle) = (\mathsf{gr}(\langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle) \ominus (b+1))$ and $\Vdash_{b+1} \langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle$ is immediate the desired result $\langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle \vdash^{\mathsf{E}}_{b+1} \texttt{unbox1}\ e : t$ follows.

In the case of `pop` we have $\Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \vdash \texttt{pop}\ e : \Box t$ because $\Gamma_0 \cdots \Gamma_b \vdash e : \Box t$. The induction hypothesis then gives $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash^{\mathsf{E}}_b e : \Box t$. Since $\mathsf{gr}(\langle \Gamma_0 \cdots \Gamma_b \rangle) = (\mathsf{gr}(\langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle) \ominus (b+1))$ and $\Vdash_{b+1} \langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle$ is immediate the desired result $\langle \Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \rangle \vdash^{\mathsf{E}}_{b+1} \texttt{pop}\ e : \Box t$ follows.

In the case of `unbox` we have $\Gamma_0 \cdots \Gamma_b \vdash \texttt{unbox}\ e : t$ because $\Gamma_0 \cdots \Gamma_b \vdash e : \Box t$. The induction hypothesis then gives $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash^{\mathsf{E}}_b e : \Box t$ and the desired result $\langle \Gamma_0 \cdots \Gamma_b \rangle \vdash^{\mathsf{E}}_b \texttt{unbox}\ e : t$ follows. □

To show that we have defined a multi-level λ-calculus we define a uniform derivor δ from L_{mlK} and L_{ml} into $\boldsymbol{\lambda}$: it simply removes all occurrences of $\Box$, `box`, `unbox1`, `pop`, and `unbox`.

It is then fairly straightforward to prove:

Fact 3.20 L_{mlK} and L_{ml} are multi-level λ-calculi.

In particular this means that both $\mathsf{MiniML}^{\Box}$ and $\mathsf{MiniML}^{\Box}_K$ can be regarded as multi-level languages despite the contradictory statement made in Subsection 5.2 of [1].

Remark about MiniML_2 of [1]. In [1] also a two-level language MiniML_2 is defined and a conservative embedding into $\mathsf{MiniML}^{\Box}_K$ is established in Theorem 4 of [1]. Since MiniML_2 is rather close to L_{cg} as defined in Subsection 3.1 we shall dispense with a formal definition of it and instead briefly comment upon a few differences. The minor differences include an explicit distinction between run-time and compile-time identifiers, a separation of the type environment into a component for run-time identifiers and one for compile-time ones, and minor differences in rules [*dn*] and [*up*]; however, despite the claim made in Subsection 4.2 of [1], both MiniML_2 and L_{cg} use the rule [*up*] for excluding run-time identifiers from the environment. The only major difference is that MiniML_2 does not faithfully model the restriction of L_{cg} that only run-time *functions* are allowed at compile-time; this seems to be *forced* by their modal approach and we regard it a rather unfortunate feature of a general framework intended to capture existing multi-level languages.

4 More general multi-level languages

So far we have considered a rather simple typed λ-calculus. To pave the way for more general definitions of multi-level languages we need to liberate ourselves from the syntax of the λ-calculus, to allow more complex type systems and even dispense with types altogether, and finally to allow additional syntactic categories. In this section we present a generalisation of multi-level λ-calculi that facilitates this; in doing so we occasionally sacrifice a bit of formality in order to maintain readability.

Programming Language. A programming language L is specified by:

- a set of sorts S^L (representing the syntactic categories); and
- a S^L-sorted signature Σ^L (representing the syntactic constructs); and
- a set $J^L = \{\vdash^{Ls} | \ s \in S^L\}$ of families of indexed well-formedness judgements $\vdash^{Ls} = (\vdash^{Ls}_i)_{i \in I^L_s}$ (on the terms of the free algebra and with I^L_s non-empty and possibly a singleton); and
- a set R^L of labelled inference rules and axioms (which we regard as inference rules with no premises); each inference rule Δ is written

$$[l] \ \frac{\cdots \vdash^{Ls_1}_{i_1} \cdots \qquad \cdots \vdash^{Ls_n}_{i_n} \cdots}{\cdots \vdash^{Ls}_{i} \cdots} \text{ if } C$$

  and its label is $lab(\Delta) = l \in N^L$, its side condition is $cond(\Delta) = C$, and its rank is $\mathsf{rank}(\Delta) = ((i_1, s_1) \cdots (i_n, s_n); (i, s))$.

We shall say that L is a programming language over S whenever $S^L = S$. We should like to stress that the above definition handles typed and untyped languages equally well.

In this definition we have been somewhat informal about the precise form of the well-formedness judgements "$\cdots \vdash^{Ls}_i \cdots$" and the side condition "C". It is counter-productive to be too formal about these points as doing so would demand a machinery that is either notationally too heavy or unnecessarily restrictive. However, we need to say a bit more in order to ensure that a derivor d from Σ^L to Σ' extends to the well-formedness judgements and the inference rules.

Concerning "$\cdots \vdash^{Ls}_i \cdots$" we anticipate that it is of form "$z_{-p}, \cdots, z_{-1} \vdash^{Ls}_i z, z_1, \cdots, z_q$" where z is a term of sort $s \in S^L$ and each z_j may be a term of some sort in S^L or constructed out of the sorts in S^L. The reason for not requiring all z_j to be terms of some sort in S^L is to allow the use of notions like type environments without having a sort for them in S^L; this favours readability over formality and we have availed ourselves of the opportunity in the previous section. Now consider a derivor d from Σ^L to Σ' that is extended to map the judgements in J^L to new judgements. This allows to define

$$d(z_{-p}, \cdots, z_{-1} \vdash^{Ls}_i z, z_1, \cdots, z_q) = d(z_{-p}), \cdots, d(z_{-1}) d(\vdash^{Ls}_i) d(z), d(z_1), \cdots, d(z_q)$$

and we shall accept that the definition is not completely formal for those z_j that do not directly correspond to sorts of S^L.

Concerning the side condition "C" we shall follow [18] and let it be built using propositional connectives ($\wedge$, $\vee$, $\Rightarrow$, $\neg$) and quantifiers ($\forall$, $\exists$) from primitive propositions; these include the well-formedness judgements "$\cdots \vdash^{Ls}_i \cdots$", sorted equality tests "$\cdots =_s \cdots$", and we shall also allow ordinary mathematical notation like function application and function update. Now consider a derivor d from Σ^L to Σ' that is extended to map the judgements in J^L to new judgements. This allows us to define

$$d\left([l] \frac{\cdots \vdash^{Ls_1}_{i_1} \cdots \qquad \cdots \vdash^{Ls_n}_{i_n} \cdots}{\cdots \vdash^{Ls}_{i} \cdots} \text{ if } C\right) \quad = \quad \frac{d(\cdots \vdash^{Ls_1}_{i_1} \cdots) \qquad d(\cdots \vdash^{Ls_n}_{i_n} \cdots)}{d(\cdots \vdash^{Ls}_{i} \cdots)} \text{ if } d(C)$$

where $d(C)$ is defined in a straightforward structural way.

We should point out the connection to equationally specified abstract data types where there is a set of (possibly conditional) equations that are used to identify given terms. In contrast we have been interested in free algebras and merely used the set of inference rules to classify terms into those that are well-formed and those that are not. These two viewpoints could be unified if we were to consider each $\vdash_i^{Ls}$ an operator in Σ^L.

A more general definition of a programming language would result if we decided to specify a set (of possibly conditional) equations between terms of a given sort; this would allow to build things like alpha-renaming of bound identifiers into the programming language.

Multi-level Structure. A multi-level structure B is given by:

- a set S^B of "sorts" (representing the syntactic categories); and
- a set W^B (also denoted B) of levels (or worlds or binding-times); and
- a $(W^B \times S^B)$-sorted signature $\Omega^B = \Omega_e^B \cup \Omega_i^B$ where Ω_i^B is given by:

  $$\Omega_i^B = \{\iota^b_{s_1 \cdots s_n ; s} \mid b \in W^B \wedge s_1, \cdots, s_n, s \in S^B\}$$

  and each operator $\iota^b_{s_1 \cdots s_n ; s}$ has rank $((b, s_1) \cdots (b, s_n); (b, s))$ and is sometimes abbreviated as ι^b_n, ι^b, or even ι.

Note that the implicit operators (in Ω_i^B) stay at the same level whereas the explicit operators (in Ω_e^B) are allowed to change between levels. We shall say that B is a multi-level structure over S whenever $S^B = S$; it is a *two-level* structure if W^B has precisely two elements and a *one-level* structure if W^B has precisely one element.

Multi-Level Language. A programming language LB is a B-level language over L iff

- there exists a set S of sorts such that L and LB are programming languages over S, and B is a multi-level structure over S; and
- the set $J^{LB} = \{\vdash^{LBs} \mid s \in S^{LB}\}$ of well-formedness judgements has $I_s^{LB} = I_s^L \times W^B$ and $\vdash^{LBs} = (\vdash_{ib}^{Ls})_{ib \in I_s^{LB}}$ whenever $\vdash^{Ls} = (\vdash_i^{Ls})_{i \in I_s^L}$; and
- there exists a function $\mathit{support} : N^{LB} \to \Omega^B$; and
- there exists a uniform derivor δ from Σ^{LB} to Σ^L that extends to a mapping from J^{LB} to J^L by setting $\delta(\vdash_{ib}^{LBs}) = \vdash_i^{Ls}$; and
- for each inference rule $\Delta \in R^{LB}$ we have:
  - there is an operator $\omega \in \Omega^B$ such that for suitable i's, b's and s's we have:

    $\mathit{support}(\mathit{lab}(\Delta)) = \omega$, and

    $\mathsf{rank}(\Delta) = (((i_1, b_1), s_1) \cdots ((i_n, b_n), s_n); ((i, b), s))$, and

    $\mathsf{rank}(\omega) = ((b_1, s_1) \cdots (b_n, s_n); (b, s))$, and

all judgements $\vdash^{LBs'}_{\iota' b'}$ occurring in $cond(\Delta)$ have[8] $b' = b$; and

– the rule $\delta(\Delta)$ is a permissible rule in L.

Note that this ensures that the derivor from LB to L maps provable judgements to provable judgements.

Homogeneous Multi-Level Language. As a special case of multi-level languages we now define the class of homogeneous multi-level languages. A programming language LB is a *homogeneous* B-level language over L if it is a B-level language over L and if it additionally satisfies:

- for all levels $b \in W^B$ the set of rules $\{\Delta \in R^{LB} \mid support(lab(\Delta)) = \iota^b\}$ is in bijective correspondence with the rules in R^L; and
- for all rules $\Delta \in R^{LB}$ such that $support(lab(\Delta)) \notin \{\iota^b \mid b \in W\}$ the premiss of the rule $\delta(\Delta)$ equals the conclusion of $\delta(\Delta)$.

Except for L_{ai} the examples considered in Section 3 are not only multi-level languages over $\boldsymbol{\lambda}$ as well as $\boldsymbol{\lambda}'$, but are indeed homogenous two-level languages over $\boldsymbol{\lambda}'$ but not $\boldsymbol{\lambda}$. The notion of B-level language defined in [13] allows B to be a partially ordered set but is somewhat more restrictive than our current notion of homogenous multi-level languages in that the above bijection is required to equal δ. Although this is sometimes the case we now believe that it is too demanding always to impose this.

5 Conclusion

In this paper we have developed a descriptive framework for multi-level λ-calculi and used it to cast further light on some of the multi-level λ-calculi found in the literature. This has had the effect of highlighting the essential differences and similarities and to pinpoint design decisions in existing calculi that should perhaps be reconsidered; examples include the restriction on fix^b in L_{pe} and L_{ml} and the "peculiar" well-typing in L_{pe} as opposed to L'_{pe}.

We have also generalised the descriptive framework so as to apply for more general classes of programming languages: we allow many more syntactic categories (for example declarations and statements), we allow more advanced typing constructs (polymorphism of one kind or the other), and we allow to dispense with types altogether.

In another direction the descriptive approach of the present paper should be complemented with a prescriptive approach as in [13]. This prescriptive approach should be more flexible than the one of [13] 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 is likely to focus on, say, the λ-calculus and seems hard to achieve for arbitrary programming languages.

[8] As previously discussed a weaker demand is that $b' \in \{b_1, \cdots, b_n, b\}$.

Acknowledgements. This work was supported in part by the DART project (The Danish Research Councils) and the LOMAPS project (ESPRIT Basic Research).

References

[1] R. Davies and F. Pfenning: A Modal Analysis of Staged Computation. *Proc. POPL'96*, pp. 258–270, ACM Press, 1996.

[2] 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.

[3] 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.

[4] F. Henglein and C. Mossin: Polymorphic Binding-Time Analysis. *ESOP'94*, Springer Lecture Notes in Computer Science, vol. 788: pp. 287–301, 1994.

[5] G. E. Hughes and M. J. Cresswell: *An Introduction to Modal Logic*. Methuen and Co. Ltd., London, 1968.

[6] 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.

[7] N. D. Jones and F. Nielson: Abstract Interpretation: a Semantics-Based Tool for Program Analysis. *Handbook of Logic in Computer Science*, vol. 4: pp. 527–636, Oxford University Press, 1995.

[8] J. C. Mitchell: Type Systems for Programming Languages. *Handbook of Theoretical Computer Science: Formal Models and Semantics*, vol. B: pp. 365–458, Elsevier Science Publishers (and MIT Press), 1990.

[9] F. Nielson: *Abstract Interpretation using Domain Theory*. PhD thesis, University of Edinburgh, Scotland, 1984.

[10] F. Nielson: Two-Level Semantics and Abstract Interpretation. *Theoretical Computer Science — Fundamental Studies*, vol. 69: pp. 117–242, 1989.

[11] F. Nielson and H. R. Nielson: Two-level semantics and code generation. *Theoretical Computer Science*, vol. 56(1): pp. 59–133, 1988.

[12] H. R. Nielson and F. Nielson: Automatic Binding Time Analysis for a Typed λ-calculus. *Science of Computer Programming*, vol. 10: pp. 139–176, 1988.

[13] F. Nielson and H. R. Nielson: *Two-Level Functional Languages*. Vol. 34 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1992.

[14] F. Nielson and H. R. Nielson: Forced Transformations of Occam Programs. *Information and Software Technology*, vol. 34(2): pp. 91–96, 1992.

[15] F. Nielson and H. R. Nielson: Multi-Level Lambda-Calculi: an Algebraic Description. *Proceedings from a Dagstuhl Seminar on Partial Evaluation*, Springer Lecture Notes in Computer Science vol. 1110: pp. 338–354, 1996.

[16] C. Stirling: Modal and Temporal Logics. *Handbook of Logic in Computer Science*, vol. 2: pp. 477–563, Oxford University Press, 1992.

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

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

A Subsets of existing systems

A.1 Code generation: [13]

In this subsection we summarise the binding time analysis of [13] (excluding product types and list types) as it pertains to the lambda calculus of the present paper.

For types [13] defines a predicate $\vdash t : b$:

$$\frac{}{\vdash \mathtt{int}^b : b} \qquad \frac{}{\vdash \mathtt{bool}^b : b} \qquad \frac{\vdash t_1 : b \quad \vdash t_2 : b}{\vdash t_1 \to^b t_2 : b} \qquad \frac{\vdash t_1 \to^r t_2 : r}{\vdash t_1 \to^r t_2 : c}$$

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

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

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

$$\frac{A \vdash e : t \to^b t : b}{A \vdash \mathtt{fix}^b\ e : t : b} \qquad \frac{A \vdash e_0 : \mathtt{bool}^b : b \quad A \vdash e_1 : t : b \quad A \vdash e_2 : t : b}{A \vdash \mathtt{if}^b\ e_0\ e_1\ e_2 : t : b}$$

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

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

A.2 Partial evaluation: [4]

In this subsection we present a restriction of the binding time analysis of [4] to the lambda calculus of the present paper. Compared with [4] we do not incorporate the qualified types (including polymorphism and constraints on binding times).

First define $\mathsf{top}(t)$ to be the annotation at the top level of t, i.e. $\mathsf{top}(\mathtt{int}^b) = b$, $\mathsf{top}(\mathtt{bool}^b) = b$, and $\mathsf{top}(t_1 \rightarrow^b t_2) = b$; in [4] one writes t^b to indicate that $\mathsf{top}(t) = b$. Then the inference system for expressions is:

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

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

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

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

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

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

A.3 Multi-level partial evaluation: [2]

In this subsection we present a restriction of the binding time analysis of [2] (expressed using Scheme) to the lambda calculus of the present paper.

For types [2] defines a predicate $\vdash t : b$:

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

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

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

For expressions the typing rules are:

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

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

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

$$\text{if } \| \ t \ \| \geq \ b$$

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

Compared with [2] we have added an obvious side condition to the rules for abstraction and lifting so as to ensure that the types derivable for the expressions are well-formed.

A.4 Modal language: [1]

In this subsection we present the modal languages $\mathsf{MiniML}^{\Box}_K$ and $\mathsf{MiniML}^{\Box}$ of [1] but ignoring the constructs for product and sum types and adding constants and conditional so as to correspond more directly to the lambda calculus of the present paper.

For types there is no notion of a well-formedness predicate. Thus all types are well-formed and we may record this by the inference rule:

$$\frac{}{\vdash t}$$

that states that all types are well-formed.

For expressions the typing rules of $\mathsf{MiniML}^{\Box}_K$ are:

$$\frac{}{\Gamma_0 \cdots \Gamma_b \vdash c_i : t} \text{ if } t = \mathsf{Type}(c_i) \qquad \frac{}{\Gamma_0 \cdots \Gamma_b \vdash x_i : t} \text{ if } t = \Gamma_b(x_i)$$

$$\frac{\Gamma_0 \cdots (\Gamma_b[x_i : t_i]) \vdash e : t}{\Gamma_0 \cdots \Gamma_b \vdash \lambda x_i.e : t_i \to t} \qquad \frac{\Gamma_0 \cdots \Gamma_b \vdash e_0 : t_1 \to t_2 \qquad \Gamma_0 \cdots \Gamma_b \vdash e_1 : t_1}{\Gamma_0 \cdots \Gamma_b \vdash e_0 \texttt{@} e_1 : t_2}$$

$$\frac{\Gamma_0 \cdots \Gamma_b \vdash e : t \to t}{\Gamma_0 \cdots \Gamma_b \vdash \texttt{fix}\ e : t} \qquad \frac{\Gamma_0 \cdots \Gamma_b \vdash e_0 : \texttt{bool} \quad \Gamma_0 \cdots \Gamma_b \vdash e_1 : t \quad \Gamma_0 \cdots \Gamma_b \vdash e_2 : t}{\Gamma_0 \cdots \Gamma_b \vdash \texttt{if}\ e_0\ e_1\ e_2 : t}$$

$$\frac{\Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \vdash e : t}{\Gamma_0 \cdots \Gamma_b \vdash \texttt{box}\ e : \Box t} \text{ if } \Gamma_{b+1} = [\,] \qquad \frac{\Gamma_0 \cdots \Gamma_b \vdash e : \Box t}{\Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \vdash \texttt{unbox1}\ e : t}$$

The system $\mathsf{MiniML}^{\Box}$ does not have the last rule but instead has the rules:

$$\frac{\Gamma_0 \cdots \Gamma_b \vdash e : \Box t}{\Gamma_0 \cdots \Gamma_b \Gamma_{b+1} \vdash \texttt{pop}\ e : \Box t} \qquad \frac{\Gamma_0 \cdots \Gamma_b \vdash e : \Box t}{\Gamma_0 \cdots \Gamma_b \vdash \texttt{unbox}\ e : t}$$

Note that when writing $\texttt{unbox1}\ e$ for $\texttt{unbox}(\texttt{pop}\ e)$ the rule for $\texttt{unbox1}$ in $\mathsf{MiniML}^{\Box}_K$ is a derived rule in $\mathsf{MiniML}^{\Box}$.