

Prescriptive Frameworks for Multi-Level Lambda-Calculi

Flemming Nielson and Hanne Riis Nielson

Computer Science Department, Aarhus University (Bldg. 540),
Ny Munkegade, DK-8000 Aarhus C, Denmark.

E-mail: {fn,hrn}@daimi.aau.dk

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. In a previous paper we have developed a *descriptive* framework for characterising the key ingredients used in the various applications. Based on the insights offered by this characterisation we now develop a *prescriptive* framework that offers firm guidelines on what we regard as “good” definitions of multi-level lambda-calculi.

1 Introduction

The literature has seen quite a number of multi-level languages used for different purposes: partial evaluation [5], code generation [8] and abstract interpretation [7]. At the surface, they all seem very different and in a recent paper [11] we performed a careful *descriptive* study of the multi-level lambda-calculi found in the literature. This allowed us to highlight a number of commonalities as well as differences between existing languages. By using a common approach we were sure not to be influenced by design decisions of no inherent importance as concerns the study of multi-level languages (like whether or not to insist that the domain of the type environment is equal to the set of free identifiers of the expression). Furthermore, we were able to identify design decisions in existing languages that should perhaps be reconsidered. In recent work [12] this descriptive approach has been extended to more general classes of programming languages than those based on

the λ -calculus; in short several syntactic categories (like expressions and commands) as well as type and kind structures can be incorporated.

Such a development is potentially very useful because it allows to “port” interesting applications like partial evaluation also to such widespread languages as C++ and Java, not to mention functional languages with primitives for concurrency and object-orientation. Yet in fulfilling this goal it becomes apparent that the descriptive approach is much too flexible and hence provides too few guidelines when addressing an unfamiliar language.

This then suggests to refine the insights obtained from the descriptive approach into something more *prescriptive*. It is natural to start with a study of λ -calculi since this is the area where multi-level languages have most flourished, and this will be the aim of the present paper. It is important to stress that while the descriptive approach aims at encompassing all design choices, the aims of the prescriptive approach is to be much more selective and indeed to exclude certain design choices.

To clarify our distinction between descriptive and prescriptive it may help to draw an analogy with denotational semantics. A main aim is the ability to *describe* a vast selection of programming languages, regardless of how intricate their semantics may seem. Another long term aim is to aid in the design of new programming languages by *prescribing* that decisions be made about a number of key notions: what are the denotable values, the storable values, the L-values etc. Indeed careful decisions at just this level was believed to be so powerful as to essentially describe the semantics of the programming language in question [13]. Thus the *prescriptive* approach aims at introducing some useful structure that captures most (but not all) of the power of the *descriptive* approach and that provides further guidance on the design of concrete applications.

In essence our prescriptive approach is based on the following key ingredients:

Permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
PEPM '97 Amsterdam, ND
© 1997 ACM 0-89791-917-3/97/0006...\$3.50

- A *multi-level structure* B containing a set B of *levels* of interest together with information about how to *pass* between them; the latter can conveniently be specified in an algebraic style by a set Ω of operators.
- A kind of *B-algebra* R specifying the conditions that have to be fulfilled in order to pass between the levels.
- A *systematic* construction of a multi-level language $L[B, R]$; the well-formedness rules for the language are naturally separated into rules belonging to a single level and into so-called *bridging rules* that allow us to move from one level to another.

This development is performed in Section 3. To facilitate obtaining a Curry-Howard isomorphism we shall annotate *all* syntactic constructs with level information and we shall have explicit syntactic constructs for passing between the levels.

While the development of Section 3 presents clear guidance concerning how to design a multi-level language it may be argued that the notation is muct too verbose: there are annotations everywhere and there is explicit syntax for passing between the levels. To illustrate the variations possible we consider in Section 4 how to introduce additional parameters to the prescriptive framework and the two main ones answer the following questions:

- Are the terms annotated with the associated level?
- Do we have explicit operators for moving between levels?

While the reader may prefer some of the variations from Section 4 over the one developed in Section 3, we believe it is important to fix the parameters “once and for all” in a manner acceptable to the community at large.

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 [3, 14] we shall represent the syntactic categories as sorts and the methods as operators.

The simply typed λ -calculus λ is the programming language specified by the following data. The *sorts* (or syntactic categories) are:

Typ, Exp

The *signature* (or the set of type and expression forming constructs) Σ is given by

$$\begin{array}{ll} \rightarrow : (\text{Typ}^2; \text{Typ}) & \text{int} : (; \text{Typ}) \\ \text{bool} : (; \text{Typ}) & \\ c_i : (; \text{Exp}) & x_i : (; \text{Exp}) \\ \lambda x_i. : (\text{Exp}; \text{Exp}) & \mathbb{0} : (\text{Exp}^2; \text{Exp}) \\ \text{if} : (\text{Exp}^3; \text{Exp}) & \text{fix} : (\text{Exp}; \text{Exp}) \end{array}$$

for i ranging over some unspecified index set.

There are two *well-formedness judgements*: $\vdash^T t$ for the well-formedness of the type t and $A \vdash^E e : t$ for the well-formedness of the expression e (yielding type t assuming that free identifiers are typed according to the type environment A). For types, we shall prefer to specify the well-formedness judgement by the following inference rules:

$$\frac{}{\vdash^T \text{int}} \quad \frac{}{\vdash^T \text{bool}} \quad \frac{\vdash^T t_1 \quad \vdash^T t_2}{\vdash^T t_1 \rightarrow t_2}$$

This is equivalent to simply stating that all types are well-formed, but as we shall see it pays off to have explicit rules.

For expressions we shall specify the well-formedness judgement by the following inference rules

$$\frac{}{A \vdash^E c_i : t} \text{ if } t = \text{Type}(c_i) \wedge \vdash^T t$$

$$\frac{}{A \vdash^E x_i : t} \text{ if } t = A(x_i) \wedge \vdash^T t$$

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

$$\frac{A \vdash^E e_0 : t_1 \rightarrow t_2 \quad A \vdash^E e_1 : t_1}{A \vdash^E e_0 \mathbb{0} e_1 : t_2}$$

$$\frac{A \vdash^E e : t \rightarrow t}{A \vdash^E \text{fix } e : t}$$

$$\frac{A \vdash^E e_0 : \text{bool} \quad A \vdash^E e_1 : t \quad A \vdash^E e_2 : t}{A \vdash^E \text{if } e_0 e_1 e_2 : t}$$

for some unspecified table Type giving the type of constants. Note that the side conditions of form $\vdash^T t$ are vacuously fulfilled, but we shall see that it pays off to include them explicitly.

In summary 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.

3 Multi-level lambda-calculi

One of the key parameters in defining a multi-level lambda calculus is the *multi-level structure* B . It is characterised by the sorts Typ and Exp inherited from λ , and

- a non-empty set W^B (also denoted B) of levels, and
- a $(W^B \times \{\text{Typ}, \text{Exp}\})$ -sorted signature Ω^B (also denoted Ω).

We shall write $|B|$ for the cardinality of W^B . In order to simplify our notation we shall assume that all operators ω have rank of the form $((b_1, s) \cdots (b_n, s); (b, s))$ for some $b_1, \dots, b_n, b \in B$ and $s \in \{\text{Typ}, \text{Exp}\}$. The multi-level structure is called *uniform* if all operators ω are of unary rank $((b_1, s); (b_2, s))$ for $b_1, b_2 \in B$ and $s \in \{\text{Typ}, \text{Exp}\}$. It turns out that most multi-level languages found in the literature fall into this subclass; the most noticeable exception being a two-level language for abstract interpretation [11, 6].

Example 3.1 As a running example throughout the paper we shall consider a (uniform) multi-level structure B_{ml} with levels $\{0, 1, \dots\}$ and the operators

$$\begin{aligned} BOX^b &: ((b+1, \text{Typ}); (b, \text{Typ})) \\ box^b &: ((b+1, \text{Exp}); (b, \text{Exp})) \\ unboxI^b &: ((b, \text{Exp}); (b+1, \text{Exp})) \end{aligned}$$

where b ranges over $\{0, 1, \dots\}$. The presence of the operator BOX^b indicates that types may be moved from level $b+1$ to the lower level b ; there is no operator allowing us to move types in the opposite direction. The presence of the operators box^b and $unboxI^b$ shows that expressions may be moved in both directions. It will emerge that this is the multi-level structure of the modal language MiniML $_K^{\square}$ [1, 11]. \square

The *multi-level λ -calculus* $L = L[B, R]$ uses the same sorts Typ and Exp as λ . The *signature* $\Sigma = \Sigma_L$ is given by

$$\begin{aligned} \rightarrow^b &: (\text{Typ}^2; \text{Typ}) & \text{int}^b &: (; \text{Typ}) \\ \text{bool}^b &: (; \text{Typ}) & & \\ c_i^b &: (; \text{Exp}) & x_i &: (; \text{Exp}) \\ \lambda^b x_i. &: (\text{Exp}; \text{Exp}) & \mathbb{Q}^b &: (\text{Exp}^2; \text{Exp}) \\ \text{if}^b &: (\text{Exp}^3; \text{Exp}) & \text{fix}^b &: (\text{Exp}; \text{Exp}) \\ \omega &: (s; s) \text{ iff } \omega \in \Omega_{((b_1, s) \cdots (b_n, s); (b, s))} \end{aligned}$$

where b ranges over B (i.e. W^B) and i ranges over some index set. Thus we have $|B|$ copies of the signature for λ where each copy is annotated with a level b from B . Furthermore, for each operator ω of rank $((b_1, s) \cdots (b_n, s); (b, s))$ we have a distinct unary operator ω in the signature for L ; these operators will be called *bridging operators*.

Well-formedness rules. The basic idea is now to have well-formedness judgements for L of the form $\vdash_b^T t$ and $A \vdash_b^E e : t$; they denote whether or not the type or expression is well-formed at level b . A well-formed term from λ that has just been annotated throughout with a particular level b is also going to be well-formed in L so the inference rules for L will contain a copy of the rules for λ for each level b in B . For types we thus have the following rules:

$$\frac{}{\vdash_b^T \text{int}^b} \quad \frac{}{\vdash_b^T \text{bool}^b} \quad \frac{\vdash_b^T t_1 \quad \vdash_b^T t_2}{\vdash_b^T t_1 \rightarrow^b t_2}$$

It is important to note that while $\vdash^T t$ holds for any type t of λ this is not going to be the case here. At this stage, if $\vdash_b^T t$ holds then all annotations of t will be b ; we shall shortly introduce rules allowing us to mix the levels in a controlled but still restricted way.

Following the same pattern we shall have $|B|$ copies of the well-formedness rules for expressions in λ :

$$\begin{aligned} \frac{}{A \vdash_b^E c_i^b : t} \text{ if } t = \text{Type}(c_i^b) \wedge \vdash_b^T t \\ \frac{}{A \vdash_b^E x_i : t} \text{ if } t = A(x_i) \wedge \vdash_b^T t \\ \frac{A[x_i : t_i] \vdash_b^E e : t}{A \vdash_b^E \lambda^b x_i. e : t_i \rightarrow^b t} \text{ if } \vdash_b^T t_i \\ \frac{A \vdash_b^E e_0 : t_1 \rightarrow^b t_2 \quad A \vdash_b^E e_1 : t_1}{A \vdash_b^E e_0 \mathbb{Q}^b e_1 : t_2} \\ \frac{A \vdash_b^E e : t \rightarrow^b t}{A \vdash_b^E \text{fix}^b e : t} \\ \frac{A \vdash_b^E e_0 : \text{bool}^b \quad A \vdash_b^E e_1 : t \quad A \vdash_b^E e_2 : t}{A \vdash_b^E \text{if}^b e_0 e_1 e_2 : t} \end{aligned}$$

It is easy to see that if we remove all annotations (or have $|B| = 1$) then we get exactly the rules for λ . Unlike Section 2 the side conditions of the form $\vdash_b^\top t$ no longer hold vacuously and hence express a deliberate restriction.

Bridging rules. So far no well-formed type or expression can contain annotations with distinct levels. To obtain some interaction between the levels we shall now introduce rules for the operators ω coming from the multi-level structure B ; these rules are called *bridging rules* since they show how to pass between the levels. The general form of these rules are as follows. For each operator $\omega_T \in \Omega$ of rank $((b_1, \text{Typ}) \cdots (b_n, \text{Typ}); (b, \text{Typ}))$ we have the rule:

$$\frac{\vdash_{b_1}^\top t_1 \cdots \vdash_{b_n}^\top t_n}{\vdash_b^\top \omega_T t} \text{ if } R_{\omega_T}(t_1, \dots, t_n, \omega_T t)$$

Here we make use of a relation R_{ω_T} to express any restrictions in the applicability of the rule; we shall later see that this relation can be viewed as the interpretation of ω_T in an algebraic structure for B . For each operator $\omega_E \in \Omega$ of rank $((b_1, \text{Exp}) \cdots (b_n, \text{Exp}); (b, \text{Exp}))$ we have the rule:

$$\frac{A_1 \vdash_{b_1}^E e_1 : t_1 \cdots A_n \vdash_{b_n}^E e_n : t_n}{A \vdash_b^E \omega_E e : t}$$

if $\vdash_b^\top t \wedge$
 $R_{\omega_E}((A_1, e_1, t_1), \dots, (A_n, e_n, t_n), (A, \omega_E e, t))$

Once more we make use of a relation R_{ω_E} to express any restrictions on the applicability of the rule.

In the formulation of the typing rules for expressions we have decided to highlight that the resulting type is well-formed as this allows us to state:

Fact 3.2 If $A \vdash_b^E e : t$ then $\vdash_b^\top t$. □

Fact 3.3 If $\vdash_b^\top t$ and $\vdash_{b'}^\top t$ then $b = b'$; if $A \vdash_b^E e : t$ and $A \vdash_{b'}^E e : t'$ then $b = b'$. □

It is easy to see that we have a Curry-Howard isomorphism: there is a one-to-one correspondance between the syntactic constructs and the inference rules of the type system.

Example 3.4 Returning to Example 3.1 we observe that the operators BOX^b , box^b and $unbox1^b$ of B_{ml} gives rise to the following operators in the signature of the multi-level language:

$$\begin{aligned} \square^b &: (\text{Typ}; \text{Typ}) \\ \text{box}^b, \text{unbox}1^b &: (\text{Exp}; \text{Exp}) \end{aligned}$$

This facilitates defining the following bridging rule for types:

$$\frac{\vdash_{b+1}^\top t}{\vdash_b^\top \square^b t}$$

It expresses that we can move a type from level $b+1$ to level b by boxing it.

For expressions we have two bridging rules:

$$\frac{A \vdash_{b+1}^E e : t}{A \vdash_b^E \text{box}^b e : \square^b t} \quad \text{if } \text{gr}(A) = \text{gr}(A)[b]$$

$$\frac{A_1 \vdash_b^E e : \square^b t}{A_2 \vdash_{b+1}^E \text{unbox}1^b e : t} \quad \text{if } \begin{aligned} \text{gr}(A_1) &= \text{gr}(A_2)[b \wedge \\ \text{gr}(A_2) &= \text{gr}(A_2)[(b+1)] \end{aligned}$$

Here

$$\begin{aligned} \text{gr}(A) &= \{(x_i^b, t) \mid A(x_i^b) = t\} \\ \text{gr}(A)[b] &= \{(x_i^{b'}, t) \in \text{gr}(A) \mid b' \leq b\} \end{aligned}$$

The rule for box^b expresses that an expression can be moved from level $b+1$ to level b by boxing the expression as well as its type. The side condition ensure that the type environment “belongs” to level b : in a judgement at level b the type environment should only contain entities that are well-formed at level b or at a lower level, i.e. the lambda-bound identifiers are all introduced at level b or lower. The rule for $\text{unbox}1^b$ expresses that an expression of boxed type can be moved from level b to level $b+1$ by unboxing it. The side condition ensures that the type environment enjoys the same properties as in the previous rule: the type environment in the conclusion should “belong” to level $b+1$ and the type environment in the premise should “belong” to level b and otherwise be equal to that of the conclusion. This allows us to prove:

Fact 3.5 If $A \vdash_b^E e : t$ and $x \in FV(e)$ then $x \in \text{dom}(A)$ and $\exists b' \leq b : \vdash_{b'}^\top A(x)$. □

The rules for \square^b , box^b and $\text{unbox}1^b$ can be reformulated so as to more directly be instances of the rule schemes considered above:

$$\frac{\vdash_{b+1}^\top t_1}{\vdash_b^\top \square^b t_2} \quad \text{if } t_1 = t_2$$

$$\frac{A_1 \vdash_{b+1}^E e_1 : t_1}{A_2 \vdash_b^E \text{box}^b e_2 : t_2} \quad \text{if } \begin{aligned} \vdash_b^\top t_2 \wedge e_1 = e_2 \wedge \\ t_2 = \text{BOX}^b t_1 \wedge \\ \text{gr}(A_1) = \text{gr}(A_2) \wedge \\ \text{gr}(A_2) = \text{gr}(A_2)[b] \end{aligned}$$

$$\frac{A_1 \vdash_b^E e_1 : t_1}{A_2 \vdash_{b+1}^E \text{unbox}1^b e_2 : t_2} \quad \text{if } \begin{aligned} \vdash_{b+1}^\top t_2 \wedge e_1 = e_2 \wedge \\ t_1 = \text{BOX}^b t_2 \wedge \\ \text{gr}(A_1) = \text{gr}(A_2)[b \wedge \\ \text{gr}(A_2) = \text{gr}(A_2)[(b+1)] \end{aligned}$$

To see that this is indeed a valid reformulation note that according to Fact 3.2 $\vdash_{b+1}^T t_1$ will hold for the rule for box^b because of its premise and hence $\vdash_b^T \square^b t_1$ will hold thanks to the rule for BOX^b and it follows that $\vdash_b^T t_2$ since $t_2 = \square^b t_1$. For the rule for $\text{unbox}1^b$ we get that $\vdash_b^T t_1$ holds from Fact 3.2 and the premise of the rule; since $t_1 = \square^b t_2$ it follows that $\vdash_{b+1}^T t_2$ holds (proved by “inversion” of the inference showing $\vdash_b^T \square^b t_2$). \square

Algebraic structure. To be more formal about the relationship between R and B we shall say that R is a B -algebra except that we interpret operators in the category \mathbf{Rel} of relations rather than in the category \mathbf{Set} of sets. In this case the carrier R_T of sort Typ consists of sets of types (i.e. terms of sort Typ) and similarly for the carrier R_E of sort Exp . For each operator ω_T of rank $((b_1, \text{Typ}) \cdots (b_n, \text{Typ}); (b, \text{Typ}))$ we shall require that R_{ω_T} specifies a $n + 1$ -ary relation on R_T . For each operator ω_E of rank $((b_1, \text{Exp}) \cdots (b_n, \text{Exp}); (b, \text{Exp}))$ we shall require that R_{ω_E} specifies a $n + 1$ -ary relation on $R_T^* \times R_E \times R_T$; this functionality is needed because R_{ω_E} is a relation over triples consisting of the type environment, the expression as well as the resulting type.

Example 3.6 The following is an algebra for the multi-level structure B_{ml} of Example 3.1. For BOX we have:

$$R_{\text{BOX}^b}(t_1, t_2) \quad \text{iff} \quad t_2 = \square^b t_1$$

For box we have:

$$\begin{aligned} & R_{\text{box}^b}((A_1, e_1, t_1), (A_2, e_2, t_2)) \\ & \quad \text{iff} \\ & e_2 = \text{box}^b e_1 \wedge t_2 = \square^b t_1 \wedge \\ & \text{gr}(A_2) = \text{gr}(A_1) \wedge \text{gr}(A_2) = \text{gr}(A_2)[b] \end{aligned}$$

For unbox we have:

$$\begin{aligned} & R_{\text{unbox}1^b}((A_1, e_1, t_1), (A_2, e_2, t_2)) \\ & \quad \text{iff} \\ & e_2 = \text{unbox}1^b e_1 \wedge t_1 = \square^b t_2 \wedge \\ & \text{gr}(A_1) = \text{gr}(A_2)[b] \wedge \text{gr}(A_2) = \text{gr}(A_2)[(b + 1)] \end{aligned}$$

It is straightforward to see that this algebra captures the formulation of Example 3.4. \square

Prescriptive versus descriptive. In a previous paper [11] we have given a general descriptive definition of a multi-level language and we shall now show that the language $L[B, R]$ introduced above is indeed a multi-level language in this sense.

In [11] we distinguish between *explicit* and *implicit* operators in the signature Ω of the multi-level structure B : there is no restriction on the ranks of the explicit

operators whereas those of the implicit operators must have the form $((s_1, b) \cdots (s_n, b); (s, b))$, i.e. they are restricted to just one level but may involve different sorts. The distinction between implicit and explicit operators is used to restrict the form of the inference rules in the multi-level language: all rules have to identify one of the operators and the premises and the conclusion of the rules have to “match” the rank of the operator. In the present development, where we take a number of copies of the rules of λ , this condition is satisfied by construction. (To be formal we just add all implicit operators in the manner of [11].)

The development of [11] also requires that we define a uniform derivor δ mapping the signature of the multi-level language to the signature of λ . In our case we shall let δ be the mapping that *deletes* all annotations and that is *undefined* on the operators ω . Note that since all ω are unary operators this does indeed define a uniform derivor in the sense of [11]. By extending δ to type environments in a pointwise manner we obtain the following result:

Fact 3.7 If $\vdash_b^T t$ then $\vdash^T \delta(t)$; if $A \vdash_b^E e : t$ then $\delta(A) \vdash^E \delta(e) : \delta(t)$. \square

showing that well-typing is preserved under erasure of annotations.

4 Variations over a theme

The framework presented in the previous section goes a long way towards capturing the essential features of the various multi-level languages presented in the literature. However, there are certain aspects that are not captured, in particular concerning the actual syntax of the multi-level language, and the extent to which the multi-level structure is reflected in the syntax:

- Are the terms annotated with the associated level?
- Do we have explicit bridging operators?

In the course of the development we shall see yet another important design decision come up:

- Does the type environment contain information about the binding level of identifiers?

So far the latter question has been of no concern thanks to the Curry-Howard isomorphism: a type can be well-formed at only one level. However, this is no longer the case when terms need not be annotated or when not all bridging operators are present. In this section we shall see how the framework can be extended to deal with these aspects and in the Conclusion we shall then discuss what constitutes a “good” design decision.

4.1 No annotation of terms

The explicit annotation of all terms with level information may seem cumbersome so we shall now present a version $L_a[B, R]$ of the multi-level language where

- a specifies which operators of the signature that have to be annotated.

Here we shall only consider the case where *none* of the operators are annotated; the previous section studied the case where *all* operators were annotated and obviously these are the two extremes of a spectrum of possibilities. The signature of $L_a[B, R]$ is then the signature of λ extended with

$$\omega : (s; s) \text{ iff } \omega \in \Omega_{((b_1, s) \dots (b_n, s); (b, s))}$$

For this approach we no longer require that there is a one-to-one correspondance between the operators ω in the multi-level structure and the operators ω in the signature of $L_a[B, R]$.

The typing judgements have the forms $\vdash_b^\top t$ and $A \vdash_b^E e : t$ as in Section 3. For types the rules are as before except that all annotations have been removed:

$$\frac{}{\vdash_b^\top \text{int}} \quad \frac{}{\vdash_b^\top \text{bool}} \quad \frac{\vdash_b^\top t_1 \quad \vdash_b^\top t_2}{\vdash_b^\top t_1 \rightarrow t_2}$$

$$\frac{\vdash_{b_1}^\top t_1 \dots \vdash_{b_n}^\top t_n}{\vdash_b^\top \omega_T t} \text{ if } R_{\omega_T}(t_1, \dots, t_n, \omega_T t)$$

Here ω_T ranges over the operators in the signature corresponding to some $\omega_T \in \Omega$ (of rank $((b_1, \text{Typ}) \dots (b_n, \text{Typ}); (b, \text{Typ}))$).

Clearly we do *not* have a Curry-Howard isomorphism for this language: for a type like `int` we will have an inference tree $\vdash_b^\top \text{int}$ for each $b \in B$. In particular this means that, contrary to what was the case in the previous section, a type can now be well-formed at many levels. Consequently it is no longer sufficient to know the type of a bound identifier in order to determine its binding level. In the rest of this subsection we shall therefore ensure that the type environment associates a type as well as a level with each identifier; we shall later consider the case where only the type is present. The typing rules for expressions then are:

$$\frac{}{A \vdash_b^E c_i : t} \text{ if } t = \text{Type}(c_i) \wedge \vdash_b^\top t$$

$$\frac{}{A \vdash_b^E x_i : t} \text{ if } t = A(x_i^b) \wedge \vdash_b^\top t$$

$$\frac{A[x_i^b : t_i] \vdash_b^E e : t}{A \vdash_b^E \lambda x_i. e : t_i \rightarrow t} \text{ if } \vdash_b^\top t_i$$

$$\frac{A \vdash_b^E e_0 : t_1 \rightarrow t_2 \quad A \vdash_b^E e_1 : t_1}{A \vdash_b^E e_0 \circ e_1 : t_2}$$

$$\frac{A \vdash_b^E e : t \rightarrow t}{A \vdash_b^E \text{fix } e : t}$$

$$\frac{A \vdash_b^E e_0 : \text{bool} \quad A \vdash_b^E e_1 : t \quad A \vdash_b^E e_2 : t}{A \vdash_b^E \text{if } e_0 e_1 e_2 : t}$$

$$\frac{A_1 \vdash_{b_1}^E e_1 : t_1 \dots A_n \vdash_{b_n}^E e_n : t_n}{A \vdash_b^E \omega_E e : t}$$

$$\text{if } \vdash_b^\top t \wedge R_{\omega_E}((A_1, e_1, t_1), \dots, (A_n, e_n, t_n), (A, \omega_E e, t))$$

Here ω_E ranges over the operators in the signature corresponding to some $\omega_E \in \Omega$ (of rank $((b_1, \text{Exp}) \dots (b_n, \text{Exp}); (b, \text{Exp}))$).

Example 4.1 An analogue of Example 3.6 for $L_a[B, R]$ is obtained by taking the signature of the multi-level language to be:

$$\begin{array}{ll} \rightarrow : (\text{Typ}^2; \text{Typ}) & \text{int} : (; \text{Typ}) \\ \text{bool} : (; \text{Typ}) & \\ c_i : (; \text{Exp}) & x_i : (; \text{Exp}) \\ \lambda x_i. : (\text{Exp}; \text{Exp}) & \circ : (\text{Exp}^2; \text{Exp}) \\ \text{if} : (\text{Exp}^3; \text{Exp}) & \text{fix} : (\text{Exp}; \text{Exp}) \\ \square : (\text{Typ}; \text{Typ}) & \text{box} : (\text{Exp}; \text{Exp}) \\ \text{unbox1} : (\text{Exp}; \text{Exp}) & \end{array}$$

By letting the type environment contain level information we obtain a multi-level language that is equal to the language L_{mlK} of [11], which is equivalent to the modal language MiniML_K^\square of [1]. \square

4.2 No bridging operators

So far we have assumed that the syntax of the multi-level language contains explicit operators for moving between the levels. We shall now present a version of the multi-level language called $L_o[B, R]$ where

- o specifies which operators from the multi-level structure B that gives rise to bridging operators.

Here we shall only consider the case where *none* of the operators give rise to bridging operators; in the previous cases *all* the operators have given rise to bridging operators and obviously these are the two extremes of a spectrum of possibilities. As our starting point in this subsection we shall take the language of Section 3 but clearly the development can be combined with that of Subsection 4.1.

The syntax of the language is now given by $|B|$ copies of the syntax of λ :

$$\begin{array}{ll} \rightarrow^b : (\text{Typ}^2; \text{Typ}) & \text{int}^b : (; \text{Typ}) \\ \text{bool}^b : (; \text{Typ}) & \\ \\ c_i^b : (; \text{Exp}) & x_i : (; \text{Exp}) \\ \lambda^b x_i. : (\text{Exp}; \text{Exp}) & \text{@}^b : (\text{Exp}^2; \text{Exp}) \\ \text{if}^b : (\text{Exp}^3; \text{Exp}) & \text{fix}^b : (\text{Exp}; \text{Exp}) \end{array}$$

For types we have $|B|$ copies of the rules for λ together with a bridging rule:

$$\frac{\vdash_{b_1}^T t_1 \cdots \vdash_{b_n}^T t_n}{\vdash_b^T t} \text{ if } R_{\omega_T}(t_1, \dots, t_n, t)$$

for $\omega_T \in \Omega$ of rank $((b_1, \text{Typ}) \cdots (b_n, \text{Typ}); (b, \text{Typ}))$.

Example 4.2 The two-level language for code generation is an example of a multi-level language of form $L_o[B, R]$. To see this let B_{cg} be the multi-level structure with levels c (for compile-time) and r (for run-time) and operators:

$$\begin{array}{l} UP : ((r, \text{Typ}); (c, \text{Typ})) \\ up : ((r, \text{Exp}); (c, \text{Exp})) \\ dn : ((c, \text{Exp}); (r, \text{Exp})) \end{array}$$

The bridging rule corresponding to UP is given by:

$$\frac{\vdash_r^T t}{\vdash_c^T t} \text{ if } \exists t_1, t_2 : t = t_1 \rightarrow^r t_2$$

The idea is that at compile-time we can manipulate run-time code (not run-time values) and since code corresponds to computations the type has to be a (run-time) function type in order to be embedded at the compile-time level. Note that a type like $\text{int}^r \rightarrow^r \text{int}^r$ will be well-formed at level r as well as c : at level r we may think of it as a run-time value (i.e. the result of a higher order computation at run-time) and at level c we may think of it as the code of some function (to be executed at run-time). \square

As in the previous subsection we clearly do not have a Curry-Howard isomorphism: a type can be well-formed at more than one level. Once more we shall therefore decide to incorporate level information into the type environment and this yields the following typing rules for expressions

$$\frac{}{A \vdash_b^E c_i^b : t} \text{ if } t = \text{Type}(c_i^b) \wedge \vdash_b^T t$$

$$\frac{}{A \vdash_b^E x_i : t} \text{ if } t = A(x_i^b) \wedge \vdash_b^T t$$

$$\frac{A[x_i^b : t_i] \vdash_b^E e : t}{A \vdash_b^E \lambda^b x_i. e : t_i \rightarrow^b t} \text{ if } \vdash_b^T t_i$$

$$\frac{A \vdash_b^E e_0 : t_1 \rightarrow^b t_2 \quad A \vdash_b^E e_1 : t_1}{A \vdash_b^E e_0 \text{@}^b e_1 : t_2}$$

$$\frac{A \vdash_b^E e : t \rightarrow^b t}{A \vdash_b^E \text{fix}^b e : t}$$

$$\frac{A \vdash_b^E e_0 : \text{bool}^b \quad A \vdash_b^E e_1 : t \quad A \vdash_b^E e_2 : t}{A \vdash_b^E \text{if}^b e_0 e_1 e_2 : t}$$

$$\frac{A_1 \vdash_{b_1}^E e_1 : t_1 \cdots A_n \vdash_{b_n}^E e_n : t_n}{A \vdash_b^E e : t}$$

$$\text{if } \frac{}{\vdash_b^T t} \wedge R_{\omega_E}((A_1, e_1, t_1), \dots, (A_n, e_n, t_n), (A, e, t))$$

where we have one copy of the latter rule for each operator $\omega_E \in \Omega$ of rank $((b_1, \text{Exp}) \cdots (b_n, \text{Exp}); (b, \text{Exp}))$.

Example 4.3 Continuing Example 4.2 the bridging rules for expressions are given by:

$$\frac{A_1 \vdash_c^E e : t}{A_2 \vdash_r^E e : t} \text{ if } \vdash_r^T t \wedge \text{gr}(A_1) \subseteq \text{gr}(A_2)$$

$$\frac{A_1 \vdash_r^E e : t}{A_2 \vdash_c^E e : t} \text{ if } \text{gr}(A_1) \subseteq \text{gr}(A_2) \wedge \text{gr}(A_2) = \text{gr}(A_2) \upharpoonright c$$

Here we define $\text{gr}(A) \upharpoonright c = \{(x_i^c, t) \in \text{gr}(A) \mid \vdash_c^T t\}$. This formulation is equal to the two-level language L_{cg} studied in [11], which is equivalent to the language considered in [10]. \square

4.3 No level information in type environment

So far we have assumed that the type environment uniquely determines the level at which the bound identifiers are introduced: In $L[B, R]$ the level could be deduced from the type since each type could be well-formed at only one level, and in $L_a[B, R]$ and $L_o[B, R]$

we ensured that the information was explicitly present in the type environment. For our final variation we introduce the parameter l where

- l specifies whether or not the type environment contains explicit level information.

Our main concern in this subsection will be the typing rules for identifiers and lambda-abstractions. In the case where all expressions are annotated with level information we simply use the following rules:

$$\frac{}{A \vdash_b^E x_i : t} \text{ if } t = A(x_i) \wedge \vdash_b^T t$$

$$\frac{A[x_i : t_i] \vdash_b^E e : t}{A \vdash_b^E \lambda x_i. e : t_i \rightarrow^b t} \text{ if } \vdash_b^T t_i$$

If the expressions are not annotated with level information the latter rule is replaced by:

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

Example 4.4 Returning to Example 4.3 let us consider the two-level language obtained by not including level information in the type environment. The relationship between the resulting two languages is very close: There is an important difference only when (i) some $A(x_i)$ has $\vdash_b^T A(x_i)$ for $b = c$ as well as $b = r$, and (ii) one cannot freely move between the possibilities $\vdash_c^T A(x_i)$ and $\vdash_r^T A(x_i)$ using the bridging rules. In fact, the language obtained here is very close to the language used in [9], and it differs from that of Example 4.3 because the bridging rules for expressions do impose constraints on type environments when moving between the levels.

Example 4.5 Finally we shall illustrate how the multi-level language developed in [2] for partial evaluation can be seen as an instance of the present framework. Consider the multi-level structure B_{pe} with levels $\{0, 1, \dots, \max\}$ and operators

$$\begin{aligned} DN_b^{b'} &: ((b + b', \text{Typ}); (b, \text{Typ})) \\ dn_b^{b'} &: ((b + b', \text{Exp}); (b, \text{Exp})) \\ up_b^{b'}, lift_b^{b'} &: ((b, \text{Exp}); (b + b', \text{Exp})) \end{aligned}$$

where $0 \leq b < b + b' \leq \max$. Here 0 is the binding level corresponding to static and $1, \dots, \max$ denote various levels of dynamic binding levels. The role of the operators $DN_b^{b'}$, $dn_b^{b'}$ and $up_b^{b'}$ are basically as in the previous examples whereas the role of $lift_b^{b'}$ will be to *lift* values at level b to level b' .

Consider now the signature obtained by annotating all the operators of λ and only adding the following bridging operator:

$$lift_b^{b'} : (\text{Exp}; \text{Exp})$$

This choice reflects that in [2] lifting is seen as the important bridging operator whereas the movement of values between levels corresponding to up and dn have lesser significance.

The bridging rule corresponding to $DN_b^{b'}$ is given by:

$$\frac{\vdash_{b+b'}^T t}{\vdash_b^T t}$$

For expressions we have the bridging rules:

$$\frac{A \vdash_b^E e : t}{A \vdash_{b+b'}^E e : t} \text{ if } \vdash_{b+b'}^T t$$

$$\frac{A \vdash_{b+b'}^E e : t}{A \vdash_b^E e : t}$$

$$\frac{A \vdash_b^E e : \text{int}^b}{A \vdash_{b+b'}^E lift_b^{b'} e : \text{int}^{b+b'}}$$

$$\frac{A \vdash_b^E e : \text{bool}^b}{A \vdash_{b+b'}^E lift_b^{b'} e : \text{bool}^{b+b'}}$$

This formulation is equivalent to the language L_{mp} presented in [11], which is equivalent to the multi-level language of [2] except that we have an operator fix^b for all choices of b whereas [2] only have the operator fix^0 , reflecting that they only see a need for static fixed point operators. In the case where $\max = 1$ this is essentially the system of [4] (excluding polymorphism). \square

5 Conclusion

Multi-level lambda-calculi. In this paper we have defined parameterised classes of multi-level lambda-calculi that capture the gist of most languages found in the literature and where the specification of the parameters give a succinct way of expressing the essentials of a given application of multi-level ideas; this follows the aims of [13] in giving succinct definitions characterising a programming language. The main parameters are:

- a multi-level structure B specifying the set of levels and operators for passing between the levels, and
- an algebraic structure R specifying the conditions that have to be fulfilled in order to pass between the levels.

The multi-level structures we have studied have a variety of properties. Some of the structures have had a finite number of levels, e.g. those for code generation [10, 9] and partial evaluation [2, 4], whereas those for modal languages have been infinite [1]. Turning to the operators of the multi-level structures they have all been unary in the present paper although examples of non-unary operators do exist (e.g. for two-level languages for abstract interpretation [11, 6]). The operators induce an ordering on the levels and in some cases it has been a reflexive ordering (e.g. for the language MiniML_K^\square of [1]) and in other cases it has been reflexive as well as transitive (e.g. for multi-level partial evaluation in [2]). We believe that the ease with which we can model such scenarios is largely a benefit of the algebraic approach; it would seem to be much more complicated to use an approach where the multi-level structure is just a partial order or even a category.

Variations over a theme. For even finer control over the multi-level language than is offered by the parameters B and R , we have given a systematic way of constructing the multi-level language and its well-formedness rules subject to the following additional parameters a , o and l :

- a specifies which operators of the signature that are to be annotated,
- o specifies which operators from the multi-level structure B that give rise to bridging operators, and
- l specifies whether or not the type environment contains explicit level information.

The table below summarises some of the main combinations considered in the paper:

	a	o	l	reference
modal lang.	none	all	yes	[1]
code gen.	all	none	yes	[10]
	all	none	no	[9]
partial eval.	all	some	no	[2, 4]

Good design choices. In the Introduction we explained the difference in motivation between a descriptive and a prescriptive approach: the *descriptive* approach aims at incorporating almost all existing systems (perhaps identifying a few design considerations in existing languages that should be reconsidered), whereas the *prescriptive* approach aims at being much more selective and indeed to exclude certain design choices.

One can view the combined development of Sections 3 and 4 as developing just one parameterised framework.

The examples then show that this is a very powerful framework that is “almost as flexible” as the descriptive approach in the ability to treat existing systems. In other words, the overall approach of this paper is not too restrictive. But perhaps it is still too flexible in not providing enough guidance when approaching an unfamiliar language?

It is therefore likely that one should choose one of the four developments (or perhaps some other less extreme combination of the possibilities we have outlined) as the canonical one. This will also allow the development of generic tools for specifying multi-level languages and for constructing parsers, inference algorithms, and other program manipulation systems. From a theoretical point of view the most pleasant candidate seems to be the system of Section 3 because it admits a Curry-Howard isomorphism. However, from a pragmatic point of view it is less clear that this is the system that “the community at large” will find most attractive since it is a rather verbose system. Only time can tell!

However, regardless of the choice of canonical framework, we believe that the way we have structured the design choices may help to increase awareness of how to design multi-level languages: by not needlessly mixing design components corresponding to “different parameters”. One analogy is the design of type systems for polymorphic languages. Here one can choose to integrate the generalisation rule with the rule for the let-construct or one can choose to keep them separate. In a similar way one can choose to integrate the instantiation rule with the rule for variables or one can choose to keep them separate. For some systems (like ML-polymorphism) the choice is non-essential, but it is usually considered “bad taste” to mix the two choices: either we integrate in both situations or else we keep the rules separate in both situations — unless of course there is a *very* good reason for deviating from this piece of advice. Another analogy is the design of operational semantics. For a given syntactic category one can choose to use a small-step semantics (also called structural operational semantics) or one can choose to use a big-step semantics (also called natural semantics). Again it is generally considered good style to use the same choice for all syntactic categories — unless of course there is a *very* good reason for deviating from this piece of advice (as indeed there occasionally is).

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] 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.
- [6] 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.
- [7] F. Nielson: Two-Level Semantics and Abstract Interpretation. *Theoretical Computer Science — Fundamental Studies*, 69: 117–242, 1989.
- [8] F. Nielson and H. R. Nielson: Two-level semantics and code generation. *Theoretical Computer Science*, 56(1): 59–133, 1988.
- [9] H. R. Nielson and F. Nielson: Automatic Binding Time Analysis for a Typed λ -calculus. *Science of Computer Programming*, 10: 139–176, 1988.
- [10] F. Nielson and H. R. Nielson: *Two-Level Functional Languages*. Vol. 34 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1992.
- [11] F. Nielson and H. R. Nielson: Multi-Level Lambda-Calculi: an Algebraic Description. *Partial Evaluation*, Springer Lecture Notes in Computer Science, vol. 1110: pp. 338–354, 1996.
- [12] F. Nielson and H. R. Nielson: Multi-Level Languages: a Descriptive Framework. Technical Report DAIMI PB-510, Computer Science Department, Aarhus University, 1996.
- [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.