Explicit Substitutitions for Constructive Necessity

Neil Ghani, Valeria de Paiva, and Eike Ritter*

School of Computer Science, University of Birmingham

This paper introduces a λ-calculus with explicit substitutions, corresponding to an S4 modal logic of constructive necessity. As well as being semantically well motivated, the calculus can be used (a) to develop abstract machines, and (b) as a framework for specifying and analysing *computation stages* in the context of functional languages. We prove several syntactic properties of this calculus, which we call xDIML, and then sketch its use as an interpretation of binding analysis and partial evaluation which respects execution of programs in stages.

1 Introduction

This paper introduces a λ-calculus with explicit substitutions, corresponding to an S4 modal logic of constructive necessity. As well as being semantically well motivated, the calculus can be used (a) to develop abstract machines, and (b) as a framework for specifying and analysing *computation stages* in the context of functional languages.

We first provide a full Curry-Howard isomorphism for the S4 constructive modal logic. This entails giving a calculus for annotating natural deduction proofs in the logic with typed λ-calculus terms. The annotations must be such that types correspond to propositions, terms correspond to proofs, and proof normalisation corresponds to term reduction. (The calculus we provide can be given a categorical model, but this is not described in this paper). We then add explicit substitutions to the calculus. Explicit substitutions are a theoretical way of making λ-calculi closer to their implementation through making substitution part of the calculus itself, rather than a meta-theoretical operation.

The first step, of providing the basic Curry-Howard isomorphism, uses the type theory developed by Barber and Plotkin [3] for Intuitionistic Linear Logic, adapting it to the modal logic by replacing the !-modality by the $\Box$-modality. We inherit the semantic foundations of Barber and Plotkin's calculus, and add to this a reduction calculus, for which we prove several syntactic properties required for developing an abstract machine.

The second step follows the approach of [8] in adding explicit substitutions to the calculus to provide the basis for an abstract machine. The addition is based on categorical techniques. Following [17] the categorical approach straightforwardly allows us to carry across, from the earlier calculus without explicit substitutions, proofs of desirable syntactic properties. Standard technology can then

* Research supported under the EPSRC project no. GR/L28296, **x-SLAM: The Explicit Substitutions Linear Abstract Machine**.

be applied to design an abstract machine from our modal calculus with explicit substitutions.

The main application of our modal calculus is as a framework for specifying and analysing *computation stages* in the context of functional languages. We use our constructive type theory to translate a variant of Nielson and Nielson's two-level functional language whose two levels correspond to compile-time and run-time (types and) programs. This gives rise to a compile-time and a run-time operational semantics. To show that this separation of types and terms into run-time and compile-time ones really works, we must show that any (closed) compile-time term c of ground type will reduce to a unique run-time term r in the compile-time operational semantics. Moreover, each closed run-time term r of ground type will reduce to a value, in the run-time operational semantics. A theorem asserting these two facts can be easily obtained from the metatheoretical results about our calculus, described in the paper.

Davies and Pfenning also discuss a modal logic and its application to binding analysis. Their operational semantics is motivated by the Kripke semantics of modal logics, whereas we use the standard operational semantics from the λ-calculus. Goubault-Larrecq [11] also presents a modal calculus with explicit substitutions, but it does not distinguish modal and intuitionistic contexts and is unsuitable for binding analysis using Nielson and Nielson's two-level functional language. Davies [5] uses a K-style modal logic to give a translation of a two-level λ-calculus into modal logic. In our approach the additional axioms of S4 are essential: the axiom $\Box A \to A$ is used for the evaluation of code at run-time.

This paper is organised as follows. First we describe a type theory for constructive modal logic S4, using a variant of Barber's Dual and Intuitionistic Linear Logic (DILL). We call this type theory Dual Intuitionistic and Modal Logic or DIML; we provide a reduction version of the calculus and sketch the proofs of the traditional syntactic requirements such as subject reduction, confluence and strong normalisation for DIML. In the next section we add explicit substitutions to DIML, obtaining xDIML. Again we discuss its reduction version and we prove that xDIML satisfies subject reduction. Then we prove the correctness of xDIML, by 'translating away' the explicit substitutions. Finally we discuss the application of our type-theory xDIML, by presenting its operational semantics and showing the preservation of execution in stages.

2 Dual Intuitionistic Modal Logic

This paper presents a proof system for constructive necessity called Dual Intuitionistic Modal Logic (DIML) which is based upon Barber's Dual Intuitionistic Linear Logic (DILL) [2] [3]. The typing judgements of DIML are of the form $\Gamma|\Delta \vdash t : A$ where Γ declares *modal variables* which may occur arbitrarily in a term, while Δ declares *intuitionistic* variables, which may not occur in any subterm of the form $\Box u$. DIML also contains *parallel* `let`-expressions of the form $\texttt{let}\ t_1, \ldots, t_n\ \texttt{be}\ \Box x_1, \ldots, \Box x_n\ \texttt{in}\ u$ which, in addition to being notationally more compact, are closer to the inherently parallel explicit substitutions of xDIML and hence simplify the correctness proof of xDIML.

2.1 The Typing Judgements of DIML

The types of DIML are ground types, function types $A \to B$ and box-types $\Box A$. Variables are tagged x_M or x_I to indicate whether they are *modal* or *intuitionistic* and the raw expressions of DIML are

$$t ::= x_M \mid x_I \mid \lambda x{:}\,A.t \mid tt \mid \Box t \mid \texttt{let}\ t_1, \ldots, t_n\ \texttt{be}\ \Box x_1, \ldots, \Box x_n\ \texttt{in}\ t$$

where the x's are variables. The tags on variables are sometimes omitted to increase legibility. We use standard abbreviations for parallel let-expressions, eg $\texttt{let}\ \bar{t}\ \texttt{be}\ \Box\overline{x}\ \texttt{in}\ v$, and identify α-equivalent terms. A context is a sequence of the form $x_1{:}\,A_1, \ldots, x_n : A_n$ where the x's are distinct variables and the A's are types. A DIML context consists of two contexts Γ and Δ containing disjoint sets of variables and is written $\Gamma|\Delta$. We call the variables in Γ *modal* and the variables in Δ *intuitionistic*. The typing judgements of DIML are of the form $\Gamma|\Delta \vdash t{:}\,A$ and are generated by the inference rules in Figure 1. Weakening for both kinds of variables is admissible because of the typing rule for these variables. Contraction of both kinds of variables follows from the additive way contexts are combined. Exchange is also admissible.

$$\Gamma, x{:}\,A, \Gamma'|\Delta \vdash x_M{:}\,A \qquad\qquad \Gamma|\Delta, x{:}\,A, \Delta' \vdash x_I{:}\,A$$

$$\frac{\Gamma|\Delta, x{:}\,A \vdash t{:}\,B}{\Gamma|\Delta \vdash \lambda x{:}\,A.t{:}\,A \to B}\ (\to I) \qquad \frac{\Gamma|\Delta \vdash t{:}\,A \to B \qquad \Gamma|\Delta \vdash u{:}\,A}{\Gamma|\Delta \vdash tu{:}\,B}\ (\to E)$$

$$\frac{\Gamma|\_ \vdash t{:}\,A}{\Gamma|\Delta \vdash \Box t{:}\,\Box A}\ (\Box I) \qquad \frac{\Gamma|\Delta \vdash t_i{:}\,\Box A_i \qquad \Gamma, x_i{:}\,A_i|\Delta \vdash u{:}\,B}{\Gamma|\Delta \vdash \texttt{let}\ t_1, \ldots, t_n\ \texttt{be}\ \Box x_1, \ldots, \Box x_n\ \texttt{in}\ u{:}\,B}\ (\Box E)$$

Fig. 1. Typing Judgements for DIML

The more complex nature of contexts in DIML results in a more subtle meta-theory. Just as there are two rules for typing variables, so there are two substitution lemmas depending on the nature of the variable being substituted. As motivation for the following lemma, consider the special case where s is $\Box x_M$.

Lemma 1 (Substitution) *If $\Gamma|\Delta \vdash t{:}\,A$ and $\Gamma|x_I{:}\,A, \Delta \vdash s{:}\,B$, then $\Gamma|\Delta \vdash s[t/x_I]{:}\,B$, where $[t/x_I]$ denotes the traditional meta-level substitution. Similarly if $\Gamma|- \vdash t{:}\,A$ and $\Gamma, x_M{:}\,A|\Delta \vdash s{:}\,B$ then $\Gamma|\Delta \vdash s[t/x_M]{:}\,B$.*

Following the approach of [7], we can regard the introduction and elimination rules as being inverse and hence derive a β and an η-equality judgement for each type constructor. This then leads to a full full Curry-Howard correspondence for DIML. However, because of space concerns we do not expand upon these remarks and instead move directly to reduction in DIML.

2.2 The reduction calculus for DIML

As is customary, reduction in DIML consists only of β-redexes and is the least congruence on raw expressions containing the basic reductions

$$\begin{array}{rcl} (\lambda x{:}\, A.t)u & \Rightarrow & t[u/x] \\ \texttt{let}\ \overline{t}, \Box u, \overline{v}\ \texttt{be}\ \Box\overline{x}, \Box y, \Box\overline{z}\ \texttt{in}\ s & \Rightarrow & \texttt{let}\ \overline{t}, \overline{v}\ \texttt{be}\ \Box\overline{x}, \Box\overline{z}\ \texttt{in}\ s[u/y] \end{array}$$

Two terms t and u which are equivalent in the equational theory generated by the DIML reduction relation are called DIML equivalent and this is written $t \cong u$. Subject reduction means that typing information is preserved by reduction and now we consider only those reductions whose redexes are well-typed terms — by subject reduction the reducts are also well-typed.

Two terms t and u which are equivalent in the equational theory generated by the DIML reduction relation are called DIML equivalent and this is written $t \cong u$. Subject reduction means that typing information is preserved by reduction and now we consider only those reductions whose redexes are well-typed terms — by subject reduction the reducts are also well-typed.

Lemma 2 (Subject Reduction) *If there is a typing judgement $\Gamma|\Delta \vdash t{:}\, A$ and a rewrite $t \Rightarrow t'$, then there is also a typing judgement $\Gamma|\Delta \vdash t'{:}\, A$.*

Theorem 3 *The relation $\Rightarrow$ is strongly normalising and confluent.*

Proof. Strong normalisation is proved by standard reducibility methods while confluence then follows from local confluence which may easily be checked.

3 Adding Explicit Substitutions to DIML

The traditional form of β-reduction $(\lambda x.t)u \Rightarrow t[u/x]$ substitutes the argument u into the body of the λ-abstraction t in one step. This is highly inefficient as each redex in u may be copied arbitrarily. Thus implementations usually evaluate terms in an environment and the contraction of a β-redex creates a new substitution which is added to the existing environment. In order to faithfully model implementations, Abadi et al. [1] proposed to make substitution part of the calculus. In this way they obtain a calculus with explicit substitution, the so-called $\lambda\sigma$-calculus, which has additional reduction rules corresponding to the inductive definition of reduction in the λ-calculus. The β-reduction of the λ-calculus is replaced by a rule which creates an explicit substitution.

We now apply this idea to DIML. In addition, just as the λ-calculus is contained in DILL, so the $\lambda\sigma$-calculus should be contained in xDIML. This containment should be such that every one-step reduction in the $\lambda\sigma$-calculus is mapped to a *one-step reduction* in xDIML. This is summarized by the following diagram in which inclusions of subcalculi are given by full lines, while the dotted lines refer to the embeddings involved in the proof that the explicit substitution calculus refines the corresponding calculus without explicit substitutions.

$$\begin{tikzcd}
\lambda\sigma\text{-calculus} \arrow[r] \arrow[d, dashed, shift right] & \text{xDIML} \arrow[d, dashed, shift right] \\
\lambda\text{-calculus} \arrow[r] \arrow[u, shift right] & \text{DIML} \arrow[u, shift right]
\end{tikzcd}$$

3.1 The Judgements of xDIML

In order to correctly add explicit substitutions to DIML we require not just terms of the form $\texttt{let}\ \overline{u}\ \texttt{be}\ \Box\overline{x}\ \texttt{in}\ t$ but also substitutions of the form $\texttt{let}\ \overline{u}\ \texttt{be}\ \Box\overline{x}\ \texttt{in}\ f$ where f is a substitution. This is equivalent to regarding the first half of a `let`-expression as an explicit substitution written $\langle\overline{u}/\Box\overline{x}\rangle$ — the above `let`-expressions are then $\langle\overline{u}/\Box\overline{x}\rangle * t$ and $\langle\overline{u}/\Box\overline{x}\rangle; f$. This replaces the notationally cumbersome `let`-expressions with "$\Box$"-substitutions whose behaviour is then governed by the $\Rightarrow_\sigma$-rewrite rules. Formally, the raw expressions of xDIML are:

$$t ::= x_I \mid x_M \mid \lambda x{:}\,A.t \mid tt \mid f * t \mid \Box t$$
$$f ::= \langle\rangle \mid \langle f, t/x_I\rangle \mid \langle f, t/x_M\rangle \mid \langle f, t/\Box x\rangle \mid f; f$$

The subscripts I and M in $\langle f, t/x_I\rangle$ and $\langle f, t/x_M\rangle$ indicate whether the substitution is for an intuitionistic or a modal variable but are sometimes omitted to increase legibility. We write $\langle t_1/x_1, \ldots, t_n/x_n\rangle$ for $\langle ..\langle\langle\rangle, t_1/x_1\rangle .., t_n/x_n\rangle$ and if $g = \langle t_1/p_1, \ldots, t_n/p_n\rangle$, we write $\langle f, g\rangle$ for $\langle f, t_1/p_1, \ldots, t_n/p_n\rangle$. A *pattern* is either x or $\Box x$, where x is a variable. A substitution $\langle t_1/p_1, \ldots, t_n/p_n\rangle$ is a *variable substitution* if each p_i is a variable pattern and a $\Box$-*substitution* if each p_i is a $\Box$-pattern. A *term substitution* is given by $s ::= \langle\rangle \mid \langle s, t/x_I\rangle, \mid \langle s, t/x_M\rangle \mid s; s$ where t is an arbitrary xDIML term. xDIML variables are bound by both λ-abstractions *and* the substitution operator $*$, *e.g.*, the variable x is bound in $\langle t/x\rangle * x$. Hence $\langle t/x\rangle * x$ is α-equivalent to $\langle t/y\rangle * y$. Consequently, the definition of bound variable is more complex than in the λ-calculus — see [16] for a full treatment. Henceforth, we identify α-equivalent terms and use Barendregt's variable convention that all bound variables are distinct from all free variables. xDIML term judgements are of the form $\Gamma|\Delta \vdash t : A$ and substitution judgements are of the form $\Gamma|\Delta \vdash f : \Gamma'|\Delta'$; these judgements are given in Figure 2. By replacing `let`-expressions by $\Box$-substitutions, we obtain an embedding of DIML into xDIML. In future, this embedding is left implicit and we regard DIML as a subcalculus of xDIML. xDIML satisfies *weakening*, *contraction* and *exchange* for both sorts of variables.

3.2 xDIML Reduction and its Correctness

Reduction in the $\lambda\sigma$-calculus consists of a modified form of β-reduction which creates an explicit substitution and a set of $\Rightarrow_\sigma$-rewrite rules which, amongst other functions, distribute substitutions over term constructors. In xDIML, such rewrite rules lead to a loss of subject reduction, eg if the rule $f * \Box u \Rightarrow_\sigma \Box f * u$ is instantiated with $f = \langle t/\Box x\rangle$ then the redex is well-typed but not the reduct.

Term Judgements

$$\Gamma, x{:}\,A, \Gamma'|\Delta \vdash x{:}\,A \qquad \Gamma|\Delta, x{:}\,A, \Delta' \vdash x{:}\,A$$

$$\frac{\Gamma|\Delta, x{:}\,A \vdash t{:}\,B}{\Gamma|\Delta \vdash \lambda x{:}\,A.t{:}\,A \to B} \quad \frac{\Gamma|\Delta \vdash t{:}\,A \to B \qquad \Gamma|\Delta \vdash u{:}\,A}{\Gamma|\Delta \vdash tu{:}\,B}$$

$$\frac{\Gamma|\ \_ \vdash t{:}\,A}{\Gamma|\Delta \vdash \Box t{:}\,\Box A} \qquad \frac{\Gamma_1|\Delta_1 \vdash f{:}\,\Gamma_2|\Delta_2 \qquad \Gamma_2|\Delta_2 \vdash t{:}\,A}{\Gamma_1|\Delta_1 \vdash f * t{:}\,A}$$

Substitution Judgements: In the last three rules, $x \notin \mathtt{dom}(\Gamma'|\Delta')$

$$\frac{\Gamma' \subseteq \Gamma \qquad \Delta' \subseteq \Delta}{\Gamma|\Delta \vdash \langle\rangle : \Gamma'|\Delta'} \qquad \frac{\Gamma_1|\Delta_1 \vdash f{:}\,\Gamma_2|\Delta_2 \quad \Gamma_2|\Delta_2 \vdash g{:}\,\Gamma_3|\Delta_3}{\Gamma_1|\Delta_1 \vdash f; g{:}\,\Gamma_3|\Delta_3}$$

$$\frac{\Gamma|\Delta \vdash f : \Gamma'|\Delta' \quad \Gamma|\_ \vdash t{:}\,A}{\Gamma|\Delta \vdash \langle f, t/x_M\rangle{:}\,\Gamma', x{:}\,A|\Delta'} \qquad \frac{\Gamma|\Delta \vdash f : \Gamma'|\Delta' \qquad \Gamma|\Delta \vdash t{:}\,A}{\Gamma|\Delta \vdash \langle f, t/x_I\rangle{:}\,\Gamma'|\Delta', x{:}\,A}$$

$$\frac{\Gamma|\Delta \vdash f : \Gamma'|\Delta' \qquad \Gamma|\Delta \vdash t{:}\,\Box A}{\Gamma|\Delta \vdash \langle f, t/\Box x\rangle{:}\,\Gamma', x : A|\Delta'}$$

Fig. 2. xDIML Typing Judgements

Thus, in xDIML, certain $\Rightarrow_\sigma$-rewrite rules only apply to *term substitutions* and new rewrite rules are added to extract term substitutions. Firstly, there is a rewrite rule which separates $\Box$-substitutions from variable substitutions — if $g = \langle t_1/p_1, \ldots, t_n/p_n\rangle$ and @ is either ; or $*$, then $g@h \Rightarrow_\sigma g_\Box @ g_V @ h$ where $g_\Box$ is g restricted to $\Box$-patterns and similarly for g_V — definitions of $g_\Box$ and g_V can be found below. If f is a $\Box$-substitution then there are also new rewrites $(f;g)@h \Rightarrow_\sigma f@(g@h)$ and $\langle f;g,t/p\rangle \Rightarrow_\sigma f;\langle g,t/p\rangle$. These rules may seem counter-intuitive as there is a rewrite in the opposite direction if f is a term substitution. However, these rewrites have a natural explanation as examples of Ghani's η-expansions for `let`-expressions [7]. Formally, xDIML reduction is the least congruence on raw expressions containing the basic redexes of Figure 3. xDIML reduction satisfies subject reduction and so we restrict ourselves to reduction on well-typed terms.

If t is a term and f a variable substitution, then there is an xDIML reduction sequence $f*t \Rightarrow_\sigma^* t[f]$ where $t[f]$ denotes the result of applying f to t. In addition, every one-step $\lambda\sigma$-rewrite is a one-step xDIML rewrite so we have the desired embedding of the $\lambda\sigma$-calculus into xDIML.

The correctness proof for the $\lambda\sigma$-calculus constructs a reduction preserving embedding of the calculus into the subcalculus containing no explicit substitutions — this embedding essentially performs all explicit substitutions. The terms of this subcalculus are λ-terms and hence normalisation and confluence for the $\lambda\sigma$-calculus follow from strong normalisation and confluence for the λ-calculus.

Modified β-reduction: $\langle f, \Box t/\Box x\rangle \Rightarrow_\beta \langle f, t/x\rangle \qquad (\lambda x{:}\, A.t)u \Rightarrow_\beta \langle u/x\rangle * t$

σ-rewrite rules: Let @ be either ; or $*$, f be a $\Box$-sub. and g a term sub.

$$
\begin{array}{llll}
\langle\rangle @ h & \Rightarrow_\sigma h & (h;\langle\rangle)@h' & \Rightarrow_\sigma h@h \\
(f;h)@h' & \Rightarrow_\sigma f@h@h' & \langle f; g, t/p\rangle & \Rightarrow_\sigma f;\langle g, t/p\rangle \\
\langle \overline{t}/\overline{p}\rangle @h & \Rightarrow_\sigma \langle \overline{t}/\overline{p}\rangle_\Box @ \langle \overline{t}/\overline{p}\rangle_V @h & & \\
 & & & \\
g@h@h' & \Rightarrow_\sigma (g;h)@h' & g;\langle h, t/p\rangle & \Rightarrow_\sigma \langle g;h,(g*t)/p\rangle \\
g*(uv) & \Rightarrow_\sigma (g*u)(g*v) & g*\lambda x.t & \Rightarrow_\sigma \lambda y.\langle g, y/x\rangle * t \ \ (y \text{ fresh}) \\
\langle g, \overline{t}/\overline{x_M}\rangle * \Box t & \Rightarrow_\sigma g*\Box(\langle \overline{t}/\overline{x_M}\rangle * t) & \langle g, s/x_I\rangle * \Box t & \Rightarrow_\sigma g*\Box t \\
\langle g, t/x\rangle * x & \Rightarrow_\sigma t & \langle g, t/y\rangle * x & \Rightarrow_\sigma g*x
\end{array}
$$

Fig. 3. Reductions for xDIML

The correctness proof for xDIML is essentially the same as we construct a reduction preserving embedding from the terms of xDIML to the terms of DIML.

An xDIML term is called *canonical* iff it is a DIML term, while an xDIML substitution is called *canonical* iff all of its sub-terms are canonical. The key clause in the embedding of xDIML into DIML is the case of a canonical substitution f applied to a canonical term t. The translation of such a term is $f \star t$ and is defined recursively by case analysis. Firstly, $\langle\rangle \star t = t$, while $f \star t = f_R \star (f_\Box * t[f_V])$ if $f_\Box \neq \langle\rangle$ and $f \star t = f_R \star t[f_V]$ if $f_\Box = \langle\rangle$. The ancillary notions $f_R, f_\Box$ and f_V are defined as follows

$$
\begin{array}{rcl}
\langle\rangle_R = \langle\rangle & \langle\rangle_\Box = \langle\rangle & \langle\rangle_V = \langle\rangle \\
\langle f, t/x\rangle_R = f_R & \langle f, t/x\rangle_\Box = f_\Box & \langle f, t/x\rangle_V = \langle f_V, t/x\rangle \\
\langle f, t/\Box y\rangle_R = f_R & \langle f, t/\Box y\rangle_\Box = \langle f_\Box, t/\Box y\rangle & \langle f, t/\Box\rangle_V = f_V \\
(f;g)_R = f; g_R & (f;g)_\Box = g_\Box & (f;g)_V = g_V
\end{array}
$$

One may easily verify that if f and t are canonical, then $f \star t$ is canonical and $f * t \Rightarrow_\sigma f \star t$. Arbitrary xDIML terms are mapped to canonical terms and arbitrary xDIML substitutions to canonical substitutions by the translation $[\![-]\!]$, which maps $f * t$ to $[\![f]\!] \star [\![t]\!]$ and respects all other term constructors. The key properties of $[\![-]\!]$ are

Proposition 4 *(i) There are reductions $f \Rightarrow^*_\sigma [\![f]\!]$ and $t \Rightarrow^*_\sigma [\![t]\!]$*
(ii) If there is a reduction $t \Rightarrow_\sigma t'$ in xDIML, then $[\![t]\!] = [\![t']\!]$.
(iii) If there is a reduction $t \Rightarrow_\beta t'$ in xDIML, then $[\![t]\!] \Rightarrow^* [\![t']\!]$.
(iv) If $t \Rightarrow t'$ in DIML, then also $t \Rightarrow^* t'$ in xDIML.

Finally we can prove confluence and normalisation:

Theorem 5 *The relation $\Rightarrow$ is confluent and every xDIML term rewrites to a unique DIML normal form.*

Proof. Given two terms u and v related in the xDIML equational theory, we have by proposition 4 that $[\![u]\!]$ and $[\![v]\!]$ are DIML equivalent. Because DIML is confluent these terms have a common reduct t and because each DIML-reduction gives rise to a sequence of xDIML-reductions both u and v reduce to t. A normal form for a term t can be computed by calculating the DIML normal form of $[\![t]\!]$ with uniqueness following from confluence.

The reduction strategy used in the normalisation proof is usually not used for abstract machines because this strategy executes all possible substitutions first and does β-reductions only afterwards. Mellies' counterexample [13] for strong normalisation of the $\lambda\sigma$-calculus applies to xDIML as well and shows that it is not strongly normalising either. Hence the termination of a particular reduction strategy needs to be shown separately. One way is to apply the reducibility method; see [18] for the proof that a wide range of reduction strategies terminate.

4 Analysis of Computation in Stages via Modal Logic

As an application of DIML we present an operational semantics for execution of functional programs in stages. Such an execution in stages is often called a *partial evaluation* of a program [12]. A program is executed in stages either because not all information is available at the same time or because we want to change the control flow of execution to execute early a part of a program that is used very often. In a functional language, this means that the variables are split into two kinds: one kind which is bound to values at compile-time, the so-called static or compile-time variables and the other kind of variables which is bound at run-time, the so-called dynamic or run-time variables. The process of separating the variables into compile-time or run-time variables is called binding analysis [4, 10, 15]. The aim of binding analysis is to identify variables as compile-time variables if they correspond to parts of the program which are guaranteed to terminate and which are arguments of needed redexes.

In this way one obtains a distinction between two kinds of function spaces: on the one hand, we have the function space $A \to B$ where the bound variable is a so-called *compile-time*-variable. Such a function is executed at compile-time. On the other hand, we have the function space $A \Rightarrow B$ where the bound variable is a so-called *run-time*-variable. Such a function is executed at run-time.

The standard presentation of this distinction uses a two-level λ-calculus [14, 15]. Such a calculus has two kinds of variables and two kinds of types. The two kinds of variables are the compile-time and run-time variables, and the types are separated into compile-time types and run-time types. Abstraction and application are defined for both compile-time types and run-time types. Redexes of compile-time type are to be reduced at compile-time, and redexes of run-time type at run-time. Run-time expressions may be encapsulated as code which the compiler may manipulate. Because normally code does not have any free variables this encapsulation must only be permitted if the run-time expression has no free run-time variables. Free compile-time variables will be bound during the compilation and are hence allowed.

There are several different versions of the two-level λ-calculus in the literature. Nielson and Nielson give only typing rules and do not specify how compile-time reductions should be performed. Palsberg defines a partial evaluator as a function who performs *all* compile-time reductions. This implies that compile-time reduction has to take place also under run-time λ-abstractions. We follow this approach as our view is that *all* compile-time reductions should be performed before any run-time execution takes place.

The distinction between compile-time and run-time has been modelled using modal logic [6]. A modal type $\Box A$ represents code of type A, and a term $\Box t$ represents a piece of code which is to be evaluated only at run-time *i.e.*, at a later stage. Davies and Pfenning also give an embedding of a two-level λ-calculus into a different modal λ-calculus which is motivated by the Kripke-semantics of modal logic where each application of the $\Box$-operator represents a new stage of the computation.

We model the execution in stages in a different way in modal logic. We use the distinction between intuitionistic and modal variables and function spaces to model the distinction between run-time and compile-time variables and function spaces respectively. The execution in stages is reflected by the separation of the operational semantics into compile-time and run-time semantics. The compile-time semantics and run-time semantics reduce only β-redexes with compile-time and run-time applications respectively. Now evaluation can proceed in stages: the compile-time operational semantics eliminates all compile-time redexes, and the run-time operational semantics eliminates all run-time redexes without re-introducing any compile-time redexes. Because in this operational semantics $\Box\Box A$ is operationally equivalent to $\Box A$, we can only model two stages of computation this way. If we want to model more stages, we have to add a new modality for each new stage. In this way we obtain the context stacks of Davies and Pfenning but with the standard operational semantics.

Our version of the two-level λ-calculus is a subcalculus of DIML. We partition the DIML-types into compile-time types C and run-time types R , given by

$$C ::= G \mid C \to^M C \mid \Box R \qquad\qquad R ::= G \mid R \to^I R$$

where $\to^I$ and $\to^M$ denote the intuitionistic and modal function space respectively, and G is any ground type. The DIML-terms are partitioned into compile-time terms c and run-time terms r with the judgements $\Gamma|\_ \vdash c\colon C$ and $\Gamma|\Delta \vdash r\colon R$, where Γ is a context of compile-time variables and Δ a context of run-time variables, which are given by the grammar

$$\begin{aligned} c &::= \texttt{const} \mid x_M \mid \lambda x^c\colon C.c \mid cc \mid \mathit{code}(r) \\ r &::= \texttt{const} \mid x_I \mid \lambda x^r\colon R.r \mid rr \mid \mathit{eval}(c) \end{aligned}$$

where `const` are the constants of ground type.

The compile-time terms stands for expressions which are evaluated at compile-time, with run-time expressions as potential values. Code is included via the constructor *code* (the DIML-term constructor $\Box$), but it is not evaluated, only copied. The run-time terms models expressions which are to be evaluated at run-time. The constructor *eval* (an abbreviation for $\texttt{let}\ x\ \texttt{be}\ \Box x\ \texttt{in}\ x$) incorporates code as produced by the compiler into the run-time system for evaluation. The term $\lambda x_M{:}\,C.c$ is an abbreviation for the DIML-term $\lambda x_I{:}\,\Box C.\texttt{let}\ x_M\ \texttt{be}\ \Box x_I\ \texttt{in}\ c$. Because our two-level λ-calculus is a subcalculus of DIML, a conservativity result in the sense of Davies and Pfenning trivially holds: a two level λ-term is well-formed if it is well-formed as a DIML-term by definition. This partitioning of terms follows the standard partitioning of DIML into intuitionistic and modal terms: A term $\Gamma|\_ \vdash t{:}\,A$ is a modal term (with only modal hypotheses), and a term $\Gamma|\Delta \vdash s{:}\,A$ is an intuitionistic term which has both intuitionistic and modal hypotheses.

Now we turn to the operational semantics. We use xDIML-expressions for the definition of the operational semantics because the step from an explicit substitution calculus to an abstract machines is small. We use the metatheory of xDIML, as developed earlier in the paper, to show the correctness of the operational semantics. The compile-time operational semantics is a call-by-value semantics for the compile-time terms together with the rule $eval(code(r)) \rightsquigarrow^c r$. We do not reduce intuitionistic applications, as these represent run-time redexes. As a consequence we must reduce under intuitionistic λ-abstractions. The values of this operational semantics of type $\Box A$ are the run-time terms with no compile-time terms as subexpressions, the values of type $C \to^M C$ are modal closures $h * \lambda x^M.t$, where h is some substitution $\langle \overline{t}/\overline{x}_M\rangle$, and the values of ground type are constants. The rules for this operational semantics are given in Figure 4.

Let h be any substitution $\langle \overline{c_v}/\overline{x}_M\rangle$ and c_v be any value.

(Var)	$\langle h, t/x_M\rangle * x_M \rightsquigarrow_c t$	$\langle h, t/y_M\rangle * x_M \rightsquigarrow_c h * x_M$
(λ_I)	$h * \lambda x_I{:}\,R.t \rightsquigarrow_c \lambda x_I{:}\,R.h * t$	$h * x_I \rightsquigarrow_c x_I$
$(\beta_\Box)$	$h * \langle f, \Box t/\Box x\rangle * x_M \rightsquigarrow_c h * t$	(β) $(h * \lambda x_M{:}\,A.t)c_v \rightsquigarrow_c \langle h, c_v/x_M\rangle * s$
(app)	$h * (ts) \rightsquigarrow_c (h * t)(h * s)$	
$(\Box)$	$h * \Box t \rightsquigarrow_c \Box h * t$	

$$\frac{s \rightsquigarrow_c s'}{ts \rightsquigarrow_c ts'} \quad \frac{t \rightsquigarrow_c t'}{tc_v \rightsquigarrow_c t'c_v} \quad \frac{t \rightsquigarrow_c t'}{\Box t \rightsquigarrow_c \Box t'}$$

plus congruence rules for all run-time constructors.

Fig. 4. Compile-time Operational Semantics

The operational semantics for the run-time terms is a standard call-by-value semantics together with the rule $eval(code(r)) \rightsquigarrow^c r$. Its values are the expected ones: constants for ground types, and intuitionistic λ-abstractions for function spaces. The rules are give in Figure 5, again for xDIML-expressions. In this way we achieve the separation of computation stages.

This staged operational semantics gives rise to a very natural abstract machines for the two-level λ-calculus: two standard abstract machines for the λ-

Let e be any substitution $\langle\overline{v}/\overline{x}_I\rangle$, and v be any value.

$$(\beta_\Box)\quad e * \langle f, \Box t/\Box x\rangle * x \rightsquigarrow_r e * t \qquad\qquad (\beta)\ (e * \lambda x_I{:}R.t)v \rightsquigarrow_r \langle e, v/x_I\rangle * t$$

$$(app)\qquad e * (ts) \rightsquigarrow_r (e * t)(e * s) \qquad\qquad \frac{s \rightsquigarrow_r s'}{ts \rightsquigarrow_r ts'}\quad \frac{t \rightsquigarrow_r t'}{tv \rightsquigarrow_r t'c_v}$$

Fig. 5. Run-time Operational Semantics

calculus; one for the modal (compile-time) β-redexes, and the other one for the intuitionistic (run-time) β-redexes. The compile-time machine needs an extension to treat run-time λ-abstractions not as values but to reduce under them. This is rather easy because the intuitionistic λ-abstraction never occurs as the function part in modal β-redexes.

The proof that the given operational semantics models the separation of execution stages relies on the meta-theoretic properties of our explicit substitution calculus xDIML. As already mentioned, we use the reducibility method applied to xDIML to show termination of the operational semantics.

Theorem 6 *(i) For each closed compile-time term c of ground type there exists a unique run-time term r such that $c \rightsquigarrow_c^* r$. Moreover, $c \Rightarrow^* r$ when we view c and r as xDIML-terms.*

(ii) For each closed run-time term r of ground type in the two-level λ-calculus there exists a unique value such that $r \rightsquigarrow_r^* v$. Moreover, $r \Rightarrow^* v$ when we view r and v as xDIML-terms.

Proof. To show the separation of stages one shows that any compile-time value of a compile-time function space is a term $\lambda x_M{:}C.c$. The uniqueness of v follows from confluence of reduction in xDIML. The existence of the value v follows from a reducibility argument similar to [18].

5 Conclusions and Further Work

This paper described a type theory xDIML associated with a modal logic for constructive necessity, that is suitable for an easy to prove correct implementation as an abstract machine. To obtain our type theory we used two processes, which seem to us generic enough to be applied to several logical systems. The first process consists of providing a Curry-Howard correspondence in the style of the dual contexts: this gave us the type theory DIML, which already satisfies most of the syntactic properties we wanted, such as subject reduction, confluence and strong normalisation. The second process is the adding of explicit substitutions to DIML to obtain xDIML. This was not as straightforward as we had hoped for, but it works and we obtain the desired syntactic results for xDIML. Is is pleasing to find that the type theory xDIML we obtain by using semantical methods (hidden way in this paper, but see [9]) can be used for practical applications such as abstract machines and binding analysis.

The implementation of the abstract machine based on this paper is the next step. This will give us means to compare different ways of doing binding analysis and partial evaluation. The semantic foundation of the machine should make it possible to give a slightly different machine for slightly different methods of binding analysis and partial evaluation, and also give a conceptual explanation of these differences.

Acknowledgements

We gratefully acknowledge discussions with Rowan Davies and Frank Pfenning about the subject of this paper.

References

1. Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jaques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. A. Barber. *Linear Type Theories, Semantics and Action Calculi*. PhD thesis, LFCS, University of Edinburgh, 1997.
3. A. Barber and G. Plotkin. Dual intuitionistic linear logic. Technical report, LFCS, University of Edinburgh, 1997.
4. C. Consel. Binding time analysis for higher order untyped functional languages. In *Proc. ACM conference on Lisp and functional programming*, pages 264–272, 1990.
5. R. Davies. A temporal logic approach to binding-time analysis. In *Proc. of LICS'96*, pages 184–193, 1996.
6. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Guy Steele, Jr., editor, *Proc. of 23rd POPL*, pages 258–270. ACM Press, 1996.
7. N. Ghani. *Adjoint Rewriting*. PhD thesis, University of Edinburgh, 1995.
8. N. Ghani, V. de Paiva, and E. Ritter. Linear explicit substitutions. In *Proc. of Westapp'98*, 1998.
9. N. Ghani, V. de Paiva, and E. Ritter. Models for explicit substitution calculi. Technical report, School of Computer Science, University of Birmingham, 1998.
10. C.K. Gomard. Partial type inference for untyped functional programs. In *Proc. ACM conference on Lisp and functional programming*, pages 282–287, 1990.
11. J. Goubault-Larrecq. Logical foundations of eval/quote mechanisms, and the modal logic S4. Manuscript, 1996.
12. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, Englewood Cliffs, NJ, 1993.
13. P.-A. Mellies. Typed λ-calculi with explicit substitution may not terminate. In *Proc. of TLCA'95*, pages 328–334. LNCS 902, 1995.
14. F. Nielson and H.R. Nielson. *Two-Level Functional Languages*. CUP, 1992.
15. J. Palsberg. Correctness of binding-time analysis. *J. of Functional Programming*, 3(3):347–363, 1993.
16. E. Ritter and V. de Paiva. On explicit substitution and names (extended abstract). In *Proc. of ICALP'97*, LNCS 1256, pages 248–258, 1997.
17. Eike Ritter. Categorical abstract machines for higher-order lambda calculi. *Theoretical Computer Science*, 136(1):125–162, 1994.
18. Eike Ritter. Normalization for typed lambda calculi with explicit substitution. In *Proc. of CSL'93*, pages 295–304. LNCS 832, 1994.