Common Object Language: A Framework for Inter-Operation Based On Higher-Order Meta-Logic

Chuck C. Liang

Department of Computer Science, Hofstra University
Hempstead, NY 11550
Email: cscccl@hofstra.edu

Abstract

This paper formulates the *common object language* framework for integrating declarative solutions into distributed and heterogeneous environments. It centers on using a common object-level representation of simply-typed lambda terms as a communication protocol. The framework is scalable between higher-order and first-order languages. It allows for the interoperation between processes in the form of remote-procedure calls, including those that require the transaction of higher-order terms. It also allows for declarative programs to invoke procedures of conventional languages such as C.

1 Introduction

Middleware [2] has emerged as an important new concept in computing. It refers to a layer of abstraction that bridges the differences between machine architectures, operating systems and programming languages. It also exists above networking protocols such as TCP/IP in that it is sensitive to high-level programming language constructs. Remote procedure call (RPC) mechanisms are examples of middleware. The growth of distributed computing in recent years has seen the introduction of several general frameworks such as CORBA, Java RMI, and Microsoft DCOM. CORBA [23] is one of the most comprehensive of these frameworks. Through CORBA, distributed programs written in possibly different languages interoperate by sharing high-level structures (classes and objects) in a client-server setting. At the heart of CORBA is an *interface definition language* (IDL) that allows the mapping of high-level structures received from a remote process to locally meaningful forms. An IDL specification forms a *universal signature* for all participating languages. It defines services that can be provided in any language and platform. We focus on this central aspect of middleware in describing the architecture of a moderate-scale, experimental CORBA-like framework suitable for use by declarative languages. The practical significance of such a framework cannot be underestimated. Declarative languages may offer superior solutions to many software problems, but no system can be expected to be best in every setting. Large, distributed software systems include components such as graphical user interfaces where declarative languages can offer little or no improvement. One can also simply wish to use software components already developed on other platforms. The ability to selectively employ declarative solutions in places where they can achieve the most effect would eliminate a serious impediment to their wide-spread adoption.

Our purpose here is not merely to adopt existing technology to a new setting. In fact, experimental mappings of CORBA IDL structures to Prolog and other declarative languages already exist (please consult Section 7 for descriptions of related work). Although our framework supports conventional as well as declarative systems, it is specifically designed to integrate logic and functional programs into distributed settings. The symbolic-computation capabilities of declarative languages will be seen as a key asset in this framework. The framework scales to support the interoperation between suitable *higher-order* languages, making possible the ability to issue, for example, *higher-order remote procedure calls.* Moreover, we demonstrate that a system for interoperation between languages can be constructed in the context of formal methods and logic.

The purpose of this paper is to describe the conceptual architecture of a general framework. The implementation of this framework is an on-going experimental project. Important fragments have already been implemented in the form of several external-language RPC mechanisms, and is available at

`http://www.cs.hofstra.edu/~cscccl/col`

Examples that can be found at this homepage involve, for example, λProlog programs calling higher-order Scheme functions, as well as C procedures invoked from several declarative languages.

2 Middleware and Meta-Logic

The encoding of *object-level* formalisms into suitable *meta-level* logical frameworks is a common practice in many areas of formal methods and logic. Interactive theorem provers such as Isabelle [17] and Coq [5] use common higher-order meta-logics to encode the syntax and semantics of various object-level logics. Meta-languages of both the functional and logic-programming variety have also been used to specify and analyze properties of object-level programming languages. These encodings involve translations of object-level structures into internal meta-level forms. For example, meta-level conjunctions are used to model object-level conjunctions in the implementation of natural deduction and sequent calculus systems. It is commonly accepted that declarative languages, especially higher-order declarative languages, are well-suited for use as meta-languages for these purposes.

The central element of the framework described here also involves the reflection of object-level structures into meta-level forms. However, here *a common object-level language is used as a medium for communication between meta-languages.*

Existing middleware frameworks such as CORBA can also be seen from this perspective. Structures of the IDL of CORBA are mapped to various equivalent structures in participating languages such as C++ and Java. However, these mappings are defined in various ad-hoc fashion. The CORBA IDL (as with similar interface languages) is also strictly first order. There is no need to worry, for instance, about normal forms. The extension of middleware to include the sharing of higher-order structures will require a much more rigorous definition of the relationship between meta- and object-level terms. The advantages of declarative languages in such a setting should be evident.

Overview

The general structure of the *common object language* (COL) framework is as follows. The COL is a formal language in its own right, with its own categories of terms, types, signatures, as well as its own typing rules. COL is a higher-order language, with a notation for lambda terms (the de Bruijn notation) suitable for efficient transmission between distributed processes. Each participating language of the framework will play the role of *meta*-language in that there will be mappings between terms and types of the COL and its internal forms. A user describes the interface to a set of shared components by writing a COL signature. *Meta-compilers* will be provided that will generate appropriate code stubs for each language participating in the framework. Distributed processes interoperate in a client-server setting through an intermediate broker process in the style of the "ORB" of CORBA.

For example, assume that a symbol declared $f : int \rightarrow int$ in the COL is to be shared between a C server and a λProlog [15] client. The meta-compilers will generate an equivalent header for C, *int f(int)*, and signature for λProlog, *type f int* $\rightarrow$ *int.* They will also generate code in each language that invoke translation procedures between COL expressions and their local equivalents. A λProlog program can therefore call for (for example) the remote evaluation of expressions such as $(f\ 3)$ without worrying about its protocol representation.

In general, we will keep in mind the following goals:

1. In keeping with the basic principle of abstraction in programming, we aim to provide maximal referential transparency for users. Translation rules between COL and local forms are to be generated and invoked automatically. Clients need not be concerned with where and with what language the services are implemented in.

2. The low-level implementations of languages should not be touched. Depending on available capabilities, languages can participate in as little or as much of the framework as appropriate. For example, a language should ideally be able to open a simple client-side socket to the intermediate broker. However, even processes that are only capable of "standard I/O" should be able to meaningfully participate in the framework via redirection services.

3. Impose a level of type safety for distributed transactions, even between untyped and unsafely-typed languages. The typing rules of the COL will be enforced by the translation between internal structures and COL terms received from a remote source.

4. Where appropriate (i.e. for well-defined declarative languages), formally establish correctness properties of the framework.

3 The Common Object Language

Conceptually speaking, the common object language is the simply-typed lambda calculus [4]. For practicality we admit the primitive types *int*, *real* and *string*. We allow a single category for "kinds:" *type*. That is, users will be able to specify new primitive type identifiers. Tuples can be derived from user declarations of appropriate constructors. Polymorphic type variables are avoided in the COL because, as will be seen, the translation between meta-level and object-level lambda terms is by induction on the structure of types. Type variables complicate this induction. However, previous work [7, 11] do show that it should be possible to extend COL to a limited, dependently-typed language, namely LF. However, simple types already suffice for many languages to interoperate and it is unclear if the dependent-type enrichment will be meaningful in practice. Furthermore, it is possible to embed LF with simply-typed terms. Experiment may indeed reveal the need for moderate extensions to the language, such as built-in pairs. In fact, it is possible to imagine extensions of this language in many directions. The language is also kept simple at this point for the purpose of experimentation.

In contrast to traditional reasons for encoding object-level languages in meta-theories, the purpose of the COL is that of communication. Inevitably, COL structures must be suitable for efficient transmission over low-level media, such as TCP/IP sockets. We therefore choose to directly specify the categories of COL expressions with this need in mind. Identifiers (or free variables) are

Kind	K ::= *type*
Types	A ::= *cint* | *creal* | *cstring* | *tid* N | *cr* A B
Terms	S ::= *cic* I | *crc* R | *csc* C | *id* N | *ind* M | *capp* S T | *cabs* S
Signatures	Γ ::= *nil* | *tid* N:K,Γ | *id* N:A,Γ | *ind* M:A,Γ

Figure 1: The Common Object Language

$$X : A, \Gamma \ \vdash_{col}\ X : A \qquad\qquad \Gamma \ \vdash_{col}\ (cic\ N) : cint$$

$$\Gamma \ \vdash_{col}\ (crc\ R) : creal \qquad\qquad \Gamma \ \vdash_{col}\ (csc\ C) : cstring$$

$$\frac{\Gamma \ \vdash_{col}\ S : A}{T : B, \Gamma \ \vdash_{col}\ S : A} \qquad\qquad \frac{ind\ 1 : A, \Gamma\uparrow \ \vdash_{col}\ S : B}{\Gamma \ \vdash_{col}\ (cabs\ S) : (cr\ A\ B)}$$

$$\frac{\Gamma \ \vdash_{col}\ S : (cr\ A\ B) \quad \Gamma \ \vdash_{col}\ T : A}{\Gamma \ \vdash_{col}\ (capp\ S\ T) : B}$$

$$(ind\ M_1 : A_1, \ldots, ind\ M_j : A_j, \Gamma)\uparrow \ = \ (ind\ M_1{+}1 : A_1, \ldots, ind\ M_j{+}1 : A_j, \Gamma)$$

Figure 2: COL Typing Rules

associated with unique non-negative integers and appropriately "tagged" as either terms or types. Integer, real, and string constants are also tagged as such. Lambda terms are represented using the de Bruijn index notation [3]. Arguments exists as to the relative merits of this representation, both in terms of user-readability and efficiency during normalization. However, our choice of this representational form is for its efficiency in representing abstractions without α-conversion. Only normal forms will be transmitted between processes, so reduction is not an issue. The nameless representation will be translated into local forms before being displayed for user-interaction. These considerations make the de Bruijn representation the logical choice for our purposes.

Formally, the COL and its type system are defined in figures 1 and 2 respectively. The symbol N represents the category of natural numbers and M that of positive numbers. I, R, and C range over integers, reals, and ASCII strings respectively. Term-level identifiers (*id N*) are distinguished from type identifiers (*tid N*). The object-level type constructor is *cr*. A *common integer constant* is tagged by the symbol *cic*. Floating point and string constants are similarly tagged. De Bruijn indices are represented by *ind M*. Application and abstraction are denoted by *capp* and *cabs* respectively.
Signatures (or contexts) are ordered lists of type and kind declarations. We further require signatures to be of the form

$$ind\ M_1 : A_1, \ldots ind\ M_j : A_j,\ id\ N_1 : B_1, \ldots id\ N_k : B_k,\ tid\ N_{k+1} : type, \ldots$$

An *initial* signature is one without declarations for de Bruijn indices. The symbol $\vdash_{col}$ represents typability relative to a signature. The operation $\uparrow$ adjusts all de Bruijn indices in a signature when abstractions are encountered. An initial signature specifies the interface of a collection of shared components (or distributed services)[1]. A term T has type A with respect to a signature Γ if $\Gamma \vdash_{col} T : A$.

At this point, rules for β and η contraction can be defined for the COL in the manner of similar calculi. They are not critical here given the intended purpose of COL. COL terms are constructed from corresponding meta-level terms. We require that all reduction be performed at the meta-level. COL terms are considered "canonical" if they are translated from meta-level terms in *$\beta\eta$-long* normal form. The long normal form is needed because the translation is defined by induction on the structure of types.

Definition 1 *A COL term is* ***canonical*** *if it is of the form*

$$cabs \ldots cabs\ (capp\ (capp\ \ldots\ (capp\ S\ T_1)\ \ldots\ T_{n-1})\ T_n)$$

where S is either a constant or de Bruijn index, n is the arity of S, and $T_1 \ldots T_n$ are also canonical. In case $n = 0$ the structure reduces to $(cabs \ldots cabs\ S)$.

Canonical COL terms are truly unique since the de Bruijn representation eliminates α-conversion. Only canonical COL terms will be transmitted between processes. This also implies that language-specific elements such as Prolog logic variables will not be encoded in the COL. Currently, types are not transmitted with terms. However, envisioned extensions of COL to dependent types will eventually require a more explicit role for types.

4 Translation Between Meta-Level and COL Terms

The actual translation between COL terms and local language forms would depend on the capabilities of the specific language in question. For certain conventional languages such as C it may only be sensible to support the first-order fragment of the framework. The code for the translation must also be suitable for automatic generation (given a signature) by a meta-compiler. In this section we define the translation between COL terms and terms of a higher-order meta-logic. The structures of this meta-logic are also simply-typed and therefore closely mirrors the terms (and types) of the COL. The main difficulty is in translating a meta-level representation into the COL de Bruijn representation. This will require a traversal over the structure of lambda terms that respects the scopes of meta-level abstractions on the one hand and correctly calculates the value of de Bruijn indices on the other. The higher-order meta-logic used to define the translation is realized in the declarative language λProlog [15], which is the principal test bed of the COL framework. However, we will show that the given translation rules are in fact generic. The simply-typed terms of the meta-logic are easily embedded in more expressive frameworks such as those based on dependent types. Additionally, it will be seen that the translation naturally scales down to the case of first-order languages such as Prolog.

The terms of the higher-order meta-logic are simply-typed lambda terms, which we shall write in the usual notation (e.g., $\lambda x \lambda y.(y\ x)$). For brevity we forgo the detailed definition of this meta-logic. Detailed definitions can be found in several sources (e.g. [14]). Signatures of the meta-logic will be denoted with Σ. We write $S^{\beta\overline{\eta}}$ for the (meta-level) $\beta\eta$-long normal form of S. The COL can be encoded in the meta-logic using the following (meta-level) signature, which we label Σ_{col}:

[1]In practical usage, users will naturally not be required to use numbers in specifying signatures. A parser will translate user specifications such as *f:int* $\rightarrow$ *int* to appropriate COL representations.

$cterm : type$ (meta-level type for COL terms)
$ctype : type$ (meta-level type for COL types)

$cint,\ creal,\ cstring,\ tid : ctype$	$cr : ctype \rightarrow ctype \rightarrow ctype$
$id,\ ind,\ cic : int \rightarrow cterm$	$crc : real \rightarrow cterm$
$csc : string \rightarrow cterm$	$cabs : cterm \rightarrow cterm$
$capp : cterm \rightarrow cterm \rightarrow cterm$	

To relate between the COL numeric representation of identifiers and meta-level identifiers, we assume the existence of a mapping $Match$, that associates COL numeric identifiers with unique meta-level (term and type) identifiers. Theoretically there are denumerably many identifiers (variables) of each type (and kind) in the meta-logic. We use f_i to denote $id\ N$ (or $tid\ N$) if $Match(f) = N$. Implicitly (not appearing as code) there is a translation between COL types and signatures and meta-level equivalents. We write $\ll A_c, A_m \gg$ to represent the correspondence between COL type A_c and meta-language type A_m. The definition of this one-to-one relation is straightforward. For example, $\ll cint, int \gg$ and $\ll cr\ A\ B, C \rightarrow D \gg$ if and only if $\ll A, C \gg$ and $\ll B, D \gg$. The full definition is omitted for brevity. We also extend this notation to denote the relationship between initial COL and meta-language signatures, whose elements are established via the $Match$ bijection.

The translation of term-level structures is given here using formulas of the higher-order meta-logic. For each meta-level type a such that $\ll A_c, a \gg$ we use a predicate symbol τ_a to write atomic formulas of the form $(\tau_a\ A_c\ S_c\ S_m\ L)$. We will assume that the "base" meta-level signature for the COL Σ_{col} also includes τ_a predicates at appropriate types. The translation is defined inductively on the structure of (COL) types. The relation

$$[\![S_c,\ T_m : A_c]\!]_l^{+/-}$$

establishes a one-to-one correspondence between COL terms S_c and meta-level terms T_m at COL type A_c. The parameter l represents an *embedding level,* which is needed to calculate de Bruijn indices. Quantifiers of the (intuitionistic) higher-order meta-logic have different meanings depending if they are on the left or right hand sides of implications ($\supset$). Following the style of [7], the relation is labeled positive ($+$) if it appears on the left of an even number of implications and negative ($-$) if appears on the left of an odd number of implications. In a logic programming context, positive clauses correspond to definite clauses while negative ones correspond to goals. We shall refer to the formulas defining this relation as *τ-clauses.*

Definition 2 *τ-clauses:*

$$[\![S,\ T : A]\!]_l^+ \ =\ [\![S,\ T : A]\!]_l^- \ =\ (\tau_a\ A\ S\ T\ l)$$
if A is a primitive type and $\ll A, a \gg$

$$[\![S,\ T : (cr\ A\ B)]\!]_l^+ \ =\ \forall u \forall v\ (\ [\![u,\ v : A]\!]_l^- \ \supset\ [\![(capp\ S\ u),\ (T\ v) : B]\!]_l^+\ )$$

$$[\![(cabs\ S),\ T : (cr\ A\ B)]\!]_l^- \ =$$
$$\forall v\ (\ (\forall m\ (m > l \supset [\![(ind\ m - l),\ v : A]\!]_m^+))\ \supset\ [\![S,\ (T\ v)^{\beta\overline{\eta}} : B]\!]_{l+1}^-\ )$$

These clauses are best implemented in λProlog and other suitably expressive higher-order declarative languages. However, *any* language can implement this translation by mimicking the operational meaning of the (intuitionistic) logical connectives, such as by generating fresh symbols for universally quantified variables. Even with languages such as C, it is possible to translate COL lambda terms into designated local data structures for processing, though the implementation of the translation would not nearly be as declarative.

A further important characteristic of this translation is that, when restricted to first-order types, the positive (definite) τ-clauses are in fact first-order Horn clauses (since $a \supset b \supset c \equiv (a \wedge b) \supset C$). The translation therefore naturally scales down to regular Prolog. For example, $\forall l \; [\![id\ 0,\ f : cr\ cint\ (cr\ cint\ cint)]\!]_l^+$ can be implemented as

```
tau(cint,capp(capp(id(0),A),C),f(B,D),L) :-
                    tau(cint,A,B,L), tau(cint,C,D,L).
```

Even the first-order restriction can provide a useful framework for many purposes.

Given appropriate positive clauses, the derivation of a negative (goal) clause will translate between well-typed COL and meta-level terms in $\beta\eta$-long normal forms. Positive τ-clauses are only given for identifiers statically and de Bruijn indices and bound variables dynamically, and thus the definition of the positive clauses need not include explicit $\beta\eta$-normalization. A normal form will always be at the "head" of a positive clause.

For each identifier in a given signature, a *meta-compiler* will generate the appropriate positive τ-clause to form a logic program stub. The concept of meta-compiling can be formalized as follows:

Definition 3 *A* ***meta-compiler*** $\mathcal{M}$ *is a function that maps an initial COL signature* Γ *to a pair consisting of a meta-level signature* Σ *such that* $\ll\Gamma, \Sigma\gg$ *and a set of* τ*-clauses of the form:*

$$\mathcal{M}(\Gamma) \;=\; (\Sigma_{col}, \Sigma;\ \{\ \forall l\ [\![f_i,\ f : A]\!]_l^+ \quad | \quad f_i : A \textit{ appears in } \Gamma\ \})$$

Here, $f_i = Match(f) = (id\ N)$ *for some* N*.*

Furthermore, $\mathcal{M}(\Gamma)$ is assumed to include $(\forall N \forall L\ \tau_{int}\ cint\ (cic\ N)\ N\ L)$, and like clauses for real and string constants. The practical implementation of $\mathcal{M}$ will also realize the mapping $Match$ from identifiers to unique numbers. Meta-compilers for non-declarative languages must emulate the functionality of $\mathcal{M}$.

It is possible to formally establish the correctness of the translation. The proofs of the following propositions are given section 9. They are in the manner of proofs of similar results in [12] (i.e., by induction on types, canonical forms, and heights of meta-logical derivations). The following proposition states that τ-clauses establish a one-to-one relation. The symbol $\vdash$ represents derivability in the higher-order meta-logic relative to a signature and a set of clauses (see [14] for definition). In effect, it is the logic programming "query" symbol `?-`.

Proposition 1 *Given an initial COL Signature* Γ *and meta-language signature* Σ *such that* $\ll\Gamma, \Sigma\gg$*, it holds that*

1. *for every canonical COL term* S *such that* $\Gamma \vdash_{col} S : A_c$*, there is a unique (up to* α*-equivalence)* Σ*-term* T *in* $\beta\eta$*-long normal form of type* A_m *such that* $\ll A_c, A_m\gg$*, and such that*

$$\mathcal{M}(\Gamma)\ \vdash\ [\![S,\ T : A_c]\!]_0^-$$

2. *and "vice versa" for every* Σ*-term* T*.*

Because τ-clauses are inductively defined on types, they serve an important purpose in addition to providing translation. In effect, they implement type checking, and under certain conditions type inference, for COL terms.

Proposition 2 *Given initial COL signature* Γ, $\mathcal{M}(\Gamma) \vdash [\![S,\ T : A]\!]_0^-$ *for some* T *if and only if* $\Gamma \vdash_{col} S : A$.

This dual-purpose of the translation clauses has important practical consequences. Through the COL framework programs of typed languages such as λProlog can interact with languages such as C. Untyped languages such as "standard" Prolog may also participate in this framework. But the COL type system exists at the object-level, and thus can be encoded even in untyped languages. The translation clauses can be used to simultaneously check the consistency of the types of COL terms received from a remote process. For example, a *cstring* constant returned by a process can be rejected by a process expecting a *cint*. This will not eliminate all unsafe programs (no type system alone will), but it does impose a level of type safety upon distributed transactions.

The τ translation allows for the implementation of referentially transparent interoperation mechanisms, such as remote evaluation for logic programming languages (see project homepage).

5 Implementing the Translation Rules

We first describe the implementation of τ-clauses in λProlog then discuss how they can be implemented in other types of programming languages.

Since λProlog embodies higher-order logic, it can naturally be expected to implement the τ translation directly. It also has certain additional features that are helpful in writing the translation. A polymorphic predicate `tau` is used for all τ_a clauses. Universal quantification in λProlog is written as `pi` and implication `=>`. Reverse implication `:-` correspond to its meaning in Prolog. These connectives can appear in both definite clauses and goals. Application is written in curried form: `(f x)` as opposed to `f(x)`. Normal forms are implicitly computed. Assuming that $Match(g) = id\,1$, the following clause is the implementation of $\forall l\ [\![id\ 1,\ g : cr\ (cr\ cint\ cint)\ cint]\!]_l^+$:

```
tau cint (capp (id 1) (cabs A)) (g B) L :-  K is (L + 1),
  pi v\ ( (pi N\ (pi M\ (tau cint (id M) v N :- M is (N - L))))
          => (tau cint A (B v) K) ).
```

The body of this positive clause is equivalent to the negative clause

$$[\![cabs\ A,\ B : cr\ cint\ cint]\!]_{l+1}^-$$

The condition $N > L$ is implicit since de Bruijn indices are never decreased. Such clauses are *reversible,* providing the translation in both directions.

Positive clauses for identifiers will be generated by the meta-compiler. Negative and positive τ-clauses converge at primitive types as atomic formulas. Additional negative clauses are needed for each type of terms that a program intends to translate. Experience in higher-order logic programming shows that this is not a problem. Lambda terms are used in higher-order programs to encode constructs such as

$$exists : (term \to formula) \to formula$$

which represents a quantifier in some encoded logic. Meta-level terms for such structures will always have primitive type *formula,* though they may embed lambda terms. In fact, most uses of higher-order programming are found to be of second order. In such a case, a single *generic* clause embodying all negative clauses would suffice[2]:

[2]This clause cannot be used at third- or higher order types because they would not translate *canonical* COL terms.

```
tau (cr S T) (cabs A) B L :-  L1 is (L + 1),
  pi v\ ( (pi N\ (pi M\ (tau S (ind M) v N :- M is (N - L))))
                  => (tau T A (B v) L1)).
```

This clause is also capable of a form of type *inference* for terms, since free logic variables (`S` and `T`) can be used to represent the COL type via unification. The derived type (such as for pure combinators) may also include free variables. They are to be understood as mere meta-level *schematic* variables for simple, primitive types. Types are not transmitted with terms, so there is no need for the COL to represent these variables.

ML and Scheme

The functional language ML naturally comes to mind when one refers to higher-order declarative languages. Indeed it is straightforward to define ML datatypes for COL terms and types. The τ-clauses naturally must be implemented separately in each direction. Furthermore, separate functions must be declared for each type (such as for `real` and `int`). Unfortunately, ML as it exists lack the ability to reflect object-level abstractions into its own native abstractions, a capability commonly referred to *higher-order abstract syntax* [18]. It would not be possible for a remote process to pass an anonymous abstraction to a higher-order ML function. COL lambda terms must be translated into structures of a designated first-order ML datatype representing object-level lambda terms. The cost of such a compromise is a loss in the degree of referential transparency. Users would need to explicitly construct underlying representations of terms before calling for (or providing) remote services.

Despite such difficulties, ML can still participate in the COL framework in meaningful ways. In fact, ML has been used effectively to manipulate object-level lambda terms in many systems. It is the implementation language of several higher-order systems such as the Isabelle theorem prover [17] and the Twelf meta-logic [19]. It is conceivable, for example, that through COL middleware a λProlog program would be able to "upload" higher-order representations of formulas into Isabelle. Future developments, such as proposed in [13, 20], may also allow ML programs to directly implement the τ-clauses at higher order.

The functional language Scheme has somewhat more flexibility in implementing the τ translation. Since Scheme is untyped, there is no need to define separate functions to produce native terms of each type. If one is willing to use features such as `eval` and `string->symbol` (as in MIT Scheme), it is in fact possible to write a translation from COL lambda terms to native Scheme functions. The core of this translation is given below:

```
(define (tau x l m)
  (cond ((eq? (car x) 'cic) (cadr x))
        ((eq? (car x) 'crc) (cadr x))
        ((eq? (car x) 'csc) (cadr x))
        ((eq? (car x) 'id)  (invmatch (cadr x)))
        ((eq? (car x) 'ind) (tolabel l m (cadr x)))
        ((eq? (car x) 'capp) (cons (tau (cadr x) l m)
                                   (list (tau (caddr x) l m))))
        ((eq? (car x) 'cabs) (let ((bv (fresh)))
              (let ((m2 (cons (cons l (list bv)) m)))
                 (cons 'lambda (cons (list bv)
                    (list (tau (cadr x) (+ l 1) m2))))))))
```

The parameter `l` for `tau` is the embedding level of abstractions while `m` is a list associating embedding levels with labels. The non-functional procedure `fresh` generates a new label each time it's called, thus mimicking $\forall$-introduction in higher-order logic. `tolabel` consults `m` to translate an index to the appropriate label, while `invmatch` is $Match^{-1}$. Calling `(tau '(cabs (ind 1)) 0 '())` will produce the expression `(lambda (v1) v1)`, which can be "`eval`'ed" into a genuine Scheme abstraction. This translation would allow a remote process to pass a lambda term to a higher-order Scheme function such as *fold.* In the reverse direction, however, it is unlikely that a function returned by a Scheme function can be translated into COL form without even more exotic features. A Scheme program can use the *quote* facility to construct a symbolic lambda term. This will again lead to a loss of transparency, though not as much as with ML.

Implementation of τ-clauses for conventional languages such as C is best restricted to the first-order fragment unless special purposes demand otherwise.

The forgoing discussion demonstrates a *hierarchy of declarativity* in languages that can implement the τ translation, and thus participate in the COL framework. Languages expressive enough to directly capture the higher-order formulas defining the τ-clauses can implement them "purely declaratively." Functional languages as they currently exist loose a degree of declarativity when implementing COL at higher order. The τ translation may be implemented directly only at the cost of using non-declarative features. Translating lambda terms into first-order data structures will decrease the level of referential transparency available to users. First-order Prolog can remain purely declarative when implementing the first-order fragment of COL (since τ-clauses are Horn clauses at first-order). Conventional languages such as C can offer powerful services to declarative systems, but are predictably the least declarative in implementing the translation, even when restricted to first order. Middleware requires the translation of high-level structures between systems. The potential advantage of declarative languages in this central aspect of middleware is thus demonstrated.

6 Communications Architecture

Once a term has been translated into COL form, a simple encoding of COL terms as a binary bit stream can be used to transmit COL terms across a low-level medium. The binary representation must preserve COL tags such as *cic* and *ind*. For example, a single byte can be used for this purpose. A rigid protocol standardization would of course need to define the binary format of COL terms carefully, including specification as to the formats of integers, floats, how strings are terminated, etc... For experimental purposes, however, such rigid standardization is unnecessary at this point. Declarative languages sometime lack binary I/O capabilities. For this reason an ASCII-string format for COL terms is also needed. This format should be easily parsable using a simple recursive-descent procedure. The syntax for COL terms was chosen with this need in mind. It is also possible to merge the invocation of τ-clause translations within a parser.

Participation in COL middleware should not require declarative languages to support elaborate network programming mechanisms such as multiple threads. The non-trivial communication mechanisms required by the COL framework are concentrated in an intermediate *broker* program. The broker maintains a registry of available services associated with the names of COL signatures. Upon receiving a client-side request, it invokes the appropriate service (if it's not already running). It will then relay communication between processes. The broker is multi-threaded to accommodate multiple communications concurrently. We currently adopt TCP stream sockets as the means for communication. Future extensions can include the usage of multiple media, such as shared memory buffers for communication between processes on the same host. The broker can be written in any programming language, such as Java, that supports its required capabilities. It is important to note that the broker does not perform any kind of processing on COL structures. It simply *relays* them.

Programs connect to the broker as clients. A *client-side* TCP socket is a relatively simple primitive that is already supported in many declarative languages, including at least one implementation of λProlog. However, even if a declarative language is only capable of "standard I/O", it should still be able to communicate with the broker via an I/O redirection program. Modern operating systems provide mechanisms to implement this kind of redirection, such as the ability to "dup" file descriptors in Unix.

At this point, it may be reasonable to ask why an existing protocol such as XML not being used to transport lambda terms. In fact, why not simply embed a notation for lambda terms inside a particular CORBA specification. These alternatives are indeed possible. However, they will lead to unnecessary overhead. Furthermore, adopting XML or CORBA as the communication format will not solve the problem of reflecting object level terms (especially lambda terms) in to meta-level terms, which is the focus of this paper. The interprocess communications protocol being used is mere implementation detail for the framework, and we have chosen a relatively simple protocol.

7 Related Work

It is common for systems to support some means for linking-in code from other languages. For example, in certain systems it is possible to call C procedures from Prolog and vice versa. Many of these *external language interfaces* are implemented in various ad hoc fashion. They are not only language-dependent, but also implementation-dependent. They often ignore the problem of how high-level structures are to be translated between languages. Some recent efforts have began to take a more principled approach, such as *H/Direct* for Haskell [8]. This mechanism adopts the use of an existing generic interface definition language to allow Haskell to interoperate with C and Microsoft COM. It is not intended to be used in a distributed setting, and is of first order.

Microsoft's .Net environment, consisting of a common *intermediate language* (.Net IL) and *common language runtime* also offers interoperability between languages. However, this system depends on proprietary, implementation-dependent features. They work by interfacing *similar* conventional languages at a lower, binary level. They are not compatible with compilers outside of the Microsoft family, even with those for the "same" languages such as C++. To participate in .Net one needs to implement a .Net-compliant compiler that must use a standard low-level data representation, as well as conform to other .Net specifications. In contrast, the system presented here (as is CORBA to a large extent) is specifically designed to support interoperation between entirely different languages *and* implementations. Only the meta-compilers, which are vastly simpler to implement than regular programming language compilers, need to be aware of specific language features. The definition of an implementation standard by a single organization does not form a general solution to how high-level structures, including those of higher order, can be translated between programming paradigms.

It is also important to point out a general difference between our work and certain efforts on common intermediate languages such as that of [21]. On the surface, such work may seem related to the intentions described here, and indeed they may be categorized as middleware in a very broad sense. However, here *"intermediate"* refers to the level between high-level languages and machine code, not between high-level structures of different languages. For example, one can conceivably compile ML into Java byte code. This would allow ML programs to run on Java virtual machines, but it does not mean that ML structures would become meaningful to a Java program. ML programs would likewise not be able to use Java classes and objects without additional interfacing mechanisms. Work in common intermediate languages are of great practical significance. It may be possible to *apply* the COL framework to pass information concerning intermediate code, such as higher-order representations of rich type structures, between platforms. Aside from this possible connection, however, common intermediate code serve different purposes from those described here. We therefore avoid using the expression "intermediate language" in describing the COL.

Much more closely related to the ideas described here are efforts such as that of [22]. A high-level interface is designed for the constraint logic programming language ECL^iPS^e. Many of the characteristics of this system are similar to those described here. The fundamental motivations for the two systems are consistent. However, the ECL^iPS^e mechanism does not address the possibility of extension to a higher-order setting.

This paper has focused on a central aspect of middleware. There are other areas, however, which remain to be addressed. Not the least of these is security. In such areas, there are also indications that formal methods and logic are beginning to be applied. The *Aster* environment for middleware described in [10] uses logical formulas to encode the requirements and behaviors of software components. At one stage, it relies on a theorem prover written in λProlog to determine the consistency between requests and services. It does not specify an interface language. Recently, middleware has also been cast in a theoretical framework of proof-net construction in linear logic [1]. Such works points to a more intelligent "broker" than the simple mechanism described here.

It is also certain that there are many other efforts in the context of formal methods and logic that are of significance to the interoperability concepts of middleware. The work in [6] is one example. As interoperability becomes of increasing interest to the declarative languages community, further advantages of declarative methods in the context of middleware will undoubtedly be revealed.

8 Conclusion

The potential advantages of higher-order declarative solutions in various domains have been well-argued in literature (see for example [16]). However, practical application usually require the integration of many diverse capabilities. The lack of interoperability with other systems has kept work on higher-order formulations in isolation. They have suffered, for example, from not being able to effectively demonstrate their usage in realistic settings. The situation is similar with many other declarative paradigms.

We have described the architecture of a generic middleware framework that scales to higher-order structures and languages. It allows for the integration of declarative solutions into large-scale, distributed, and heterogeneous systems. Furthermore, it is designed to not require special changes to the implementations of languages. In the process, we have also reconstructed a fundamental fragment of middleware as a concept that can be well-defined in logic. Middleware offers a new form of abstraction in programming. Declarative languages not only can participate, but can offer clear advantages in this new paradigm.

9 Proofs

Proposition 1 is stated for initial signatures only, for which the meta-compiler will generate a static program stub. The computation of the τ clauses, however, will involve the dynamic introduction of new clauses. The proof of proposition 1 is therefore given for a more general result.

In the higher-order meta-logic, proving an universally quantified formula amounts to proving that it holds for a fresh *eigenvariable* not appearing free elsewhere. The (meta-level) signature is thus dynamically expanded by these constants. We shall use Π to denote a *dynamic* meta-level signature of the form

$$v_1 : B_1, \ \ldots, \ v_n : B_n$$

where each v_i is a unique variable not appearing free elsewhere. From a given Π, we also define its COL equivalent and its manifestation as formulas dynamically introduced during the derivation of τ-clauses as follows:

Definition 4

Given meta-level dynamic signature $\Pi = (v_1 : B_1, \ \ldots, \ v_n : B_n)$

1. $\Delta^{\Pi} \ = \ (ind\ 1 : A_1, \ \ldots, \ ind\ n : A_n)$
   where $\ll A_i, B_i \gg$ *for* $1 \leq i \leq n$.

2. $D_0^{\Pi} = \{ \ \}$

   $D_n^{\Pi} = \{\forall m \ (m > n–i) \ \supset \ [\![ind\ m–(n–i), \ v_{i+1} : A_{i+1}]\!]_m^{+} \ \mid \ 1 \leq i \leq n\}$

3. $D_n^{\Pi} \uplus \mathcal{M}(\Gamma) = \Pi, \Sigma; D_n^{\Pi} \cup \mathcal{G} \quad$ *(if* $\mathcal{M}(\Gamma) = \Sigma; \mathcal{G}$*).*

The generic result from which proposition 1 immediately follows is then:

Theorem 1 *Given initial COL Signature* Γ *and meta-language signature* Σ *such that* $\ll\Gamma, \Sigma\gg$*, it holds that*

1. *For arbitrary dynamic signature* $\Pi = (v_1 : B_1, \ \ldots, \ v_n : B_n)$*, and for every canonical COL term* S *such that* $\Delta^{\Pi}, \Gamma \ \vdash_{col} \ S : A$*, there is a unique (up to* α*-equivalence)* Π, Σ*-term* T *in* $\beta\eta$*-long normal form such that*

$$D_n^{\Pi} \uplus \mathcal{M}(\Gamma) \ \vdash \ [\![S, \ T : A]\!]_n^{-}$$

2. *and "vice versa" for every* Π, Σ*-term* T*.*

Proof:

The proofs of the two directions are analogous. We focus on the forward direction. The proof is by induction on the structure of S (or equivalently on the structure of type A), which is assumed to be in canonical form.

Base Case: if S is an integer, real or string constant, the result follows from the fact that $\mathcal{M}(\Gamma)$ contains $\forall N \forall L(\tau_{int}\ cint\ (cic\ N)\ N\ L)$ and analogous clauses for reals and strings. We fold the proof of the case where S is a zero-arity constant or de Bruijn index into the more general case of applications.

Inductive Case: we prove separately the cases where S is and is not an abstraction.

S is not an abstraction:
In this case, since S is in canonical form, A must be of primitive type. We will show that for a unique T,

$$D_n^\Pi \uplus \mathcal{M}(\Gamma) \ \vdash \ \tau_a \ A \ S \ T \ n$$

where S is of the form

$$S \ = \ capp \ \ldots \ (capp \ F \ S_1) \ \ldots \ S_j$$

Here, F is a non-compound term of type $(cr \ A_1 \ \ldots \ (cr \ A_j \ A) \ldots)$ (analogously $a_1 \to \ldots \to a_j \to a$ at the meta-level), since S is assumed to be well-typed. We assume $j \geq 0$ to include the case of zero-arity constants and indices. In each of the following cases we show how to construct a (backwards) proof:

1. F is an identifier $f_i = Match(f) = (id \ K)$ for some f and K. In this case, $\mathcal{M}(\Gamma)$ contains

   $$\forall m [\![f_i,\ f : (cr \ A_1 \ \ldots (cr \ A_j \ A) \ldots)]\!]^+_m,$$

   which is equal to

   $$\begin{gathered}\forall m \ (\ \forall u_1 \forall w_1 [\![u_1,\ w_1 : A_1]\!]^-_m \ \supset \ \ldots \ \supset \ \forall u_j \forall w_j [\![u_j,\ w_j : A_j]\!]^-_m \\ \supset \ \tau_a \ A \ (capp \ldots (capp \ f_i \ u_1) \ldots u_j) \ (f \ w_1 \ldots w_j) \ m \ )\end{gathered}$$

   f exits uniquely since $Match$ is bijective. By definition of canonical forms, each S_i for $1 \leq i \leq j$ is also in canonical form. Thus by inductive hypothesis we have that for $1 \leq i \leq j$,

   $$D_n^\Pi \uplus \mathcal{M}(\Gamma) \ \vdash \ [\![S_i,\ T_i : A_i]\!]^-_n$$

   for unique $T_1 \ldots T_j$ in $\beta\eta$-long normal form. We can construct a (backwards) proof that

   $$D_n^\Pi \uplus \mathcal{M}(\Gamma) \ \vdash \ \tau_a \ A \ S \ T \ n$$

   by a reverse application of $\forall$-left ($\forall L$) which substitutes m with n, followed by a series of j analogous steps involving pairs of $\forall L$ and $\supset L$. We use the following abreviations:

   - $\mathcal{P} = D_n^\Pi \uplus \mathcal{M}(\Gamma)$
   - $C = (\tau_a \ A \ S \ T \ n)$, and
   - $C_1 = \tau_a \ A \ (capp \ldots (capp \ f_i \ S_1) \ldots u_j) \ (f \ T_1 \ldots w_j) \ n$,

   In addition, by $G + \mathcal{P}$ where G is a formula and $\mathcal{P}$ a signature and a set of formulas, we mean the extension of the set of formulas in $\mathcal{P}$ by G. The structure of the proof is as follows:

   $$\dfrac{\dfrac{\mathcal{P} \vdash [\![S_1,\ T_1 : A_1]\!]^-_n \qquad \dfrac{\dfrac{\mathcal{P} \vdash [\![S_j,\ T_j : A_j]\!]^-_n \qquad C + \mathcal{P} \vdash C}{\vdots}\ \supset L}{\forall u_2 \forall w_2 [\![u_2,\ w_2 : A_2]\!]^-_n \supset \ldots \supset C_1 + \mathcal{P} \vdash C}\ \forall L\ \forall L}{\dfrac{[\![S_1,\ T_1 : A_1]\!]^-_n \supset \forall u_2 \forall w_2 [\![u_2,\ w_2 : A_2]\!]^-_n \supset \ldots \supset C_1 + \mathcal{P} \vdash C}{\dfrac{[\![f_i,\ f : (cr \ A_1 \ \ldots (cr \ A_j \ A) \ldots)]\!]^+_n + \mathcal{P} \vdash C}{\forall m [\![f_i,\ f : (cr \ A_1 \ \ldots (cr \ A_j \ A) \ldots)]\!]^+_m + \mathcal{P} \vdash C}\ \forall L}\ \forall L\ \forall L}\ \supset L$$

   Let $T = (f \ T_1 \ldots T_j)$, which is unique and in normal form since $T_1 \ldots T_n$ are. The result then follows.

2. F is an index $(ind\ I)$. In this case, since S is well-typed with respect to Δ^{Π}, Γ, it can be shown that $1 \leq I \leq n$. There is thus a unique clause in D_n^{Π} of the form

$$\forall m\ (m > n - I) \supset [\![ind\ m\text{–}(n\text{–}I),\ v_{n\text{-}I+1} : (cr\ A_1\ \ldots (cr\ A_j\ A) \ldots)]\!]_m^+$$

By $\forall L$ we can again substitute m with n and get

$$(n > n\text{–}I) \supset [\![ind\ I,\ v_{n\text{-}I+1} : (cr\ A_1\ \ldots (cr\ A_j\ A) \ldots)]\!]_n^+$$

The precondition $(n > n\text{–}I)$ can be dropped since $I \geq 1$. The rest of the proof for this case can be constructed as in the previous case for identifiers.

This completes the case where S is not an abstraction (and consequently for primitive types A).

S is an abstraction:
this is also the case where A is a non-primitive type $(cr\ A_0\ B_0)$. Since S is canonical and well-typed, it must be of the form $(cabs\ S')$ where

$$ind\ 1 : A_0, \Delta^{\Pi}\uparrow, \Gamma\ \vdash_{col}\ S' : B_0$$

We show that for some T',

$$D_n^{\Pi} \uplus \mathcal{M}(\Gamma)\ \vdash\ \forall v(\ (\forall m(m > n \supset [\![ind\ m-n,\ v : A_0]\!]_m^+))\ \supset\ [\![S',\ (T'\ v)^{\beta\overline{\eta}} : B_0]\!]_{n+1}^-$$

By a reverse application of $\forall$-right, we substitute v with a fresh variable v_0. Let $\Pi' = v_0 : a_0, \Pi$ (where $\ll A_0, a_0 \gg$) Now by a reverse application of $\supset$-right, we reduce the proof to that of

$$D_{n+1}^{\Pi'} \uplus \mathcal{M}(\Gamma)\ \vdash [\![S',\ (T'\ v_0)^{\beta\overline{\eta}} : B_0]\!]_{n+1}^-$$

which follows by inductive hypothesis on the subterm S'. Furthermore, the existence of $(T'\ v_0)^{\beta\overline{\eta}}$ is unique upto α-equivalence. So let $T = \lambda v_0.(T'\ v_0)^{\beta\overline{\eta}}$, and we thus have that

$$D_n^{\Pi} \uplus \mathcal{M}(\Gamma)\ \vdash\ [\![S,\ T : A]\!]_n^-$$

□

One direction of proposition 2 follows from the above theorem. To prove the other direction we again generalize the result.

Theorem 2 *Given initial COL signature Γ, meta-level signature Π, Σ such that $\ll \Gamma, \Sigma \gg$, if $D_n^{\Pi} \uplus \mathcal{M}(\Gamma) \vdash [\![S,\ T : A]\!]_n^-$ for some T then $\Delta^{\Pi}, \Gamma\ \vdash_{col}\ S : A$.*

Proof:

The proof is by induction on the height of *uniform* proofs, which again mirrors the structure of S and A.

For the base case where $D_n^{\Pi} \uplus \mathcal{M}(\Gamma) \vdash [\![S,\ T : A]\!]_n^-$ in one step, the result follows directly from the definitions of D_n^{Π} and $\mathcal{M}(\Gamma)$.

In the inductive case, if the last step of the (meta-level) proof is an application of **Backchain**, then it must be that

$$S = capp \ldots (capp\ F\ S_1) \ldots S_j$$

where F is a non-compound term, and $D^{\Pi}_n \uplus \mathcal{M}(\Gamma)$ contains a clause $\forall m[\![F,\ G : A]\!]^+_m$ (with an additional condition $m > n - l$ in case F is an index). This cluase is of the form

$$\forall m\ (\ \forall u_1 \forall w_1 [\![u_1,\ w_1 : A_1]\!]^-_m\ \supset\ \ldots\ \supset\ \forall u_j \forall w_j [\![u_j,\ w_j : A_j]\!]^-_m$$

$$\supset\ \tau_a\ A\ (capp \ldots (capp\ F\ u_1) \ldots u_j)\ (G\ w_1 \ldots w_j)\ m\ )$$

From the assumption of provability we know that

$$D^{\Pi}_n \uplus \mathcal{M}(\Gamma) \vdash [\![S_i,\ T_i : A_i]\!]^-_n$$

for some T_i and A_i, which by inductive hypothesis means that $\Delta^{\Pi}, \Gamma\ \vdash_{col}\ S_i : A_i$. A series of applications of the typing rule for *capp* will then derive a type judgement for $capp \ldots (capp\ F\ S_1) \ldots S_j$.

If the last step of the proof is not backchain, then it must be a **Generic**, preceded by a **Augment**. That is, we have that

$$\dfrac{\dfrac{D^{\Pi'}_{n+1} \uplus \mathcal{M}(\Gamma)\ \vdash\ [\![S',\ (T\ v)^{\beta\overline{\eta}} : B_0]\!]^-_{n+1}}{\vdots}\ \textbf{Aug}}{D^{\Pi}_n \uplus \mathcal{M}(\Gamma)\ \vdash\ \forall v(\ (\forall m(m > n \supset [\![ind\ m{-}n,\ v : A_0]\!]^+_m))\ \supset\ [\![S',\ (T\ v)^{\beta\overline{\eta}} : B_0]\!]^-_{n+1})}\ \textbf{Gen}$$

where $S = cabs\ S'$ and $\Pi' = v_0 : a_0, \Pi$ for some fresh variable v_0. By inductive hypothesis, it follows that

$$\Delta^{\Pi'}, \Gamma\ \vdash_{col}\ S' : B_0$$

where $\Delta^{\Pi'} = ind\ 1, \Delta^{\Pi}\uparrow$. And thus

$$\Delta^{\Pi}, \Gamma\ \vdash_{col}\ (cabs\ S') : A_0 \to B_0$$

□

References

[1] J. M. Andreoli. Focussing proof-net construction as a middleware paradigm. In *18th International Conference on Automated Deduction*, pages 501–516. Springer-Verlag LNAI no. 2392, 2002.

[2] P. A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[3] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.

[4] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[5] G. Dowek et al. The Coq proof assistant user's guide. Technical Report 134, INRIA, 1993, 1993.

[6] G. Gupta et al. Semantics-based filtering: Logic programming's killer app? In *4th International Symposium on Practical Aspects of Declarative Languages*, pages 82–100. Springer-Verlag LNCS no. 2257, 2002.

[7] A. Felty. Encoding dependent types in an intuitionistic logic. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[8] S. Finne, D. Leijen, E. Meijer, and S. P. Jones. *H/Direct:* A Binary Foreign Language Interface for Haskell. In *Proceedings of the International Conference on Functional Programming*, 1998.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[10] V. Issarny, C. Bidan, and T. Saridakis. Achieving midleware customization in a configuration-based development evironment: Experience with the Aster prototype. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 207–214, 1998.

[11] C. Liang. Substitutions for proofs and types as logic programming. In *Proceedings of the Workshop on Proof Search in Type Theoretic Languages.* Part of the 13th International Conference on Automated Deduction, Rutgers University, 1996.

[12] C. Liang. *Substitution, Unification and Generalization in Meta-Logic.* PhD thesis, University of Pennsylvania, December 1995.

[13] D. Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, June 1990.

[14] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[15] G. Nadathur and D. Miller. An overview of λProlog. In K. A. Bowen and R. A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, August 1988.

[16] G. Nadathur and D. Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 499–590. Oxford University Press, 1998.

[17] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[18] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[19] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction*, pages 202–206. Springer-Verlag LNAI 1632, 1999.

[20] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Programme Construction, 5th International Conference*. Lecture Notes in Computer Science, no. 1837, 2000.

[21] Z. Shao. Typed common intermediate format. In *1997 USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, Oct 1997.

[22] K. Shen, J. Schimpf, S. Novello, and J. Singer. A High-Level Generic Interface to External Programming Languages for ECLiPSe. In *4th International Symposium on Practical Aspects of Declarative Languages*, pages 262–279. Springer-Verlag LNCS no. 2257, 2002.

[23] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications*, 14(2), 1997.