Modular Logic Metaprogramming

Karl Klose
Aarhus University
Denmark

Klaus Ostermann
University of Marburg
Germany

Abstract

In logic metaprogramming, programs are not stored as plain textfiles but rather derived from a deductive database. While the benefits of this approach for metaprogramming are obvious, its incompatibility with separate checking limits its applicability to large-scale projects. We analyze the problems inhibiting separate checking and propose a class of logics that reconcile logic metaprogramming and separate checking. We have formalized the resulting module system and have proven the soundness of separate checking. We validate its feasibility by presenting the design and implementation of a specific logic that is able to express many metaprogramming examples from the literature.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.4 [*Programming Languages*]: Processors: Code generation

General Terms Design, Languages, Theory

Keywords Modularity, Separate Checking, Logic Metaprogramming

1. Introduction

The need to divide large-scale software into manageable building blocks became apparent in the 1960s and 1970s. It was soon recognized, however, that not every partition of the source code into blocks constitutes a good modularization; rather, a good module should be self-contained in that it describes its requirements on the context and clearly specifies the service it provides to the external world in a form that is separately checkable by a compiler while still hiding implementation details [35]. Dedicated module constructs that help to enforce such abstraction boundaries quickly followed and have been realized in the module constructs of languages such as Ada, Pascal, C, Modula-2, or ML. Indeed, Cardelli (and others) have argued that separate checking is such a fundamental property of a module system that a module system without separate checking is (arguably) not really a modularization mechanism [8].

Separate checking is traditionally achieved by specifying a set of names or signatures of functions, procedures, or data-structures that the module requires (or *imports*) and provides (or *exports*), respectively. A module can be checked separately by checking that the module provides well-formed definitions of all exported names under the assumption that definitions for all imported names will be provided by other modules.

However, this simple notion of import/export interfaces as lists of names or signatures breaks down once some form of (static) metaprogramming is involved. Static metaprogramming means that (parts of) the program are *computed* at compile-time. Typical examples of static metaprogramming are templates in C++, various forms of macro programming, or compile-time metaobject protocols such as OpenC++ [11] or Javassist [12]. If such program computations are involved in the compilation process, it is neither clear how these program computations can be checked separately (meaning that they will only compute well-formed programs for every well-formed input), nor how the functionality of these computations can be described as an interface that does not reveal the implementation details, such that clients that use the computed programs can be checked against this interface. As a consequence, programs involving static metaprograms can usually only be understood, analyzed and verified as a whole, because the dependencies between parts of such programs cannot be specified in terms of module interfaces. This problem is well-known, and various solutions have been proposed for specific restricted forms of metaprogramming, such as conditional declarations [26] or certain computations over the structure of classes [19, 25, 27, 28].

This work has a more general focus: We investigate and characterize the relation between separate checking and metaprogramming, independent of specific metaprogramming constructs or host languages. We identify minimal conditions on the interface and implementation language under which separate checking is sound.

We have chosen *logic metaprogramming* [14, 30, 39] as the basis for our studies, because it is one of the most general and declarative forms of metaprogramming, and because the strong semantic foundations of logic programming make it particularly well-suited for a formal treatment. In logic metaprogramming, programs are described as a deductive database in a logic language such as Prolog, and thus the shape of the final program may depend on arbitrarily complex deduction rules.

Although we use logic metaprogramming as the motivation of our approach, we believe that our results are also applicable to other forms of metaprogramming, or, more generally, to any form of programming that requires modules whose interfaces require some form of computation or deduction.

The contributions of this work are as follows:

- We analyze the tension between logic metaprogramming and separate checking and, more generally, between program generators, information hiding, and separate checking of well-formedness constraints.
- We propose a generic model of modules that describes the implementation and interface of modules in terms of formulas from a logic language, such that compatibility and consistency checks can be expressed in terms of logic validity and consistency. This model does not depend on a fixed logic or object language; rather, it axiomatizes the requirements on the logic and the object language under which separate checking is sound. Module constructs ranging from traditional modules which import/export lists of names to sophisticated modules that can give interfaces to program generators can be uniformly described as instances of this model, via logics that satisfy the axioms. The structure of our formalization has been inspired by (and can be seen as a generalization of) Cardelli's seminal work on separate checking [8].
- We present a design and implementation of an actual module description logic, $\mathcal{F}_{P+}$, that fits to the axioms described in our formalization and is sufficiently powerful for many typical use cases from logic metaprogramming. The logic is the pure part of Prolog (no negation-as-failure, cuts etc.) plus a few simple, well-known extensions from λProlog [33]. We demonstrate by means of examples how a small, Java-like object-oriented programming language can be modeled in our system and be equipped with metaprogramming constructs.
- We explore the design space of instantiations of our model by presenting other possible logic languages and discussing their respective expressiveness to model object languages and metaprogramming constructs. We also discuss applications of modular logic metaprogramming for generating design pattern implementations, cross-language type systems, and pluggable type systems.

The remainder of this paper is structured as follows. First, we introduce logic metaprogramming and discuss its modularity problems (Sec. 2). Then we show informally what steps need to be taken to make logic metaprogramming modular (Sec. 3). In Sec. 4, we present our formalization of module description logics. We describe an implementation of a module system based on the formalization and the implementation of the $\mathcal{F}_{P+}$ logic in Sec. 5. We discuss advantages and limitations of our approach in Sec. 6. Section 7 compares to related work. Section 8 concludes. Appendix A contains the proofs of all theorems presented in the paper.

2. Logic Metaprogramming

In (static) logic metaprogramming (LMP) [14, 39], a logic language (the *meta language*) is used to specify programs in another language (the *object language*). In the following we give a brief introduction to LMP by encoding a typical metaprogramming example using Prolog as meta-language and a functional toy object language which we call FP. We will discuss both the expressiveness and the problems of LMP in terms of this example.

Although the main implementation of our framework is for the Java programming language, we will use FP as an example object language in this section. Since it is much simpler, it allows us to show every aspect of modeling a language in our framework for modular logic metaprogramming (MLMP) in full detail. Although Java is a much more powerful and complex language, it does not add many interesting issues for our purposes that do not already arise in FP. We will discuss our Java incarnation of our model in Sec. 5.3.

2.1 Logic Metaprogramming Systems

An LMP system consists of three parts: A front-end, a deductive database, and a back-end.

The *front-end* reads object language terms from data streams, and converts these terms to a form understood by the meta language. For example, the front-end could translate a function `fun f = arg+1`[1] into the Prolog term `funimpl(f,add(arg,1))`. For our further treatment we ignore the front-end, since it is not interesting, and concentrate on the part where the input is already transformed into logic formulas.

The *deductive database* contains the representation of the object language program generated by the front-end and a set of program computation rules. The program computation rules are either in some form encoded in the object language programs, or the programmer may directly access the deductive database, or there could be just a fixed set of predefined computation rules. The origin of the program computation rules is irrelevant for the purpose of this paper. The database

[1] FP is a first-order functional language with integers and addition as only available primitive type and operation, respectively. Each function has exactly one implicit parameter which is called `arg`.

```
funimpl(f1, add(5, arg)).
funimpl(f2, add(7, call(f1, arg))).

funimpl(compose(A, B),
       add(call(A, arg), call(B, arg)))
  ← funimpl(A, _), funimpl(B, _).

funimpl(f3, add(arg, call(compose(f1,f2), 13))).
```

Figure 1. LMP Example

may also contain auxiliary (library) rules that help in writing program computation rules, such as rules to look up things in the program. Furthermore, the deductive database contains a deduction mechanism, which can be used to derive all possible variable substitutions under which a query can be deduced from the database.

The *back-end* of the system uses the deductive database to generate the program in object language syntax. To this end, the back-end typically generates all solutions to queries that represent object language entities of interest for the current program, and generates code for these solutions. For example, the query `funimpl(Name, Body)` generates all function definitions that can be deduced from the current database. For each successful substitution found for these queries code is generated. If the database contains recursively applicable program computation rules, the naïve strategy of iterating over all possible program entities will not terminate. To solve the problem of such infinitely large code bases, a strategy to select only those parts of the deductive closure of the database that are relevant to the program at hand is required. This could be just a list of function names, or a computation of every program entity transitively referenced from the "main" function. The choice of this strategy is irrelevant for this work; we just assume that some terminating program generation strategy is available.

2.2 Example: A Generic Function Composer

The example in Fig. 1 shows a typical use case for metaprogramming. The code defines two ordinary functions `f1` and `f2`, and a function defined by a proper metaprogram: `compose` composes two functions `A` and `B` by calling each with the given argument and adding their results. [2] The last function `f3` uses the metaprogram by calling a composed function `compose(f1, f2)`. Using a straight-forward semantics for FP, we expect `f3` to be equivalent to the function `add(arg, 43)` at runtime.

This small metaprogram is a typical example of the use of metaprogramming to solve a programming problem that requires abstraction over elements of the language that are not 'first class': In our example language FP it is not possible to write abstractions involving functions, but we can provide such abstractions using metaprogramming.

Although metaprogramming would not be necessary to implement this example in a higher-order functional language, every programming language has some structures that are not 'first class', so metaprograms are needed to implement abstractions over these structures. For example, classes in object-oriented programming languages like Java or C# or data type definitions in Haskell are not 'first class'. The lack of abstractions over classes, in particular, is a commonly encountered problem and metaprogramming techniques like code generation and language extensions are often used to deal with it. For example, the definition of generic proxy classes or pair classes is a typical metaprogramming example in OO languages [25].

Our example can be seen as a small-scale version of these more sophisticated applications of metaprogramming. We have implemented examples for the aforementioned more realistic usages of metaprogramming, but they are more complicated and do not add anything to the discussion that cannot also be illustrated in terms the simplistic function generator example.

Since metaprograms cannot be understood by the compiler or interpreter for the object language, the system needs to generate code for the instance of `compose` that is used in `f3`. How the resulting names for composite functions such as `compose(f1, f2)` are represented in object code is a responsibility of the back-end. For example, it could generate a new function name such as `compose_f1_f2`.

2.3 Representation Choices

There are several design choices in the representation of object programs. For our purposes, the granularity and the openness of the representation are significant. Regarding granularity, function bodies could also be treated and processed as opaque constants rather than being decomposed into term representations of their abstract syntax trees as in our example. Only in the latter case can the deductive database also be used to perform interesting computations on the expression level, but if such kinds of computations are not required, then an opaque constant representation is sufficient.

Representations also differ in their openness, which boils down to the decision whether a constructor in the object language is represented as a term constructor (closed) or predicate constructor (open) in its encoding. We have represented function definitions via a predicate, which means that new functions can be added to the database without changing any existing code. Function bodies, on the other hand, are represented as terms, which makes it impossible to add something to a function body without actually modifying the term. This difference becomes interesting when one considers, for example, whether the methods of an object-oriented class should be represented in an open or a closed way. These choices have implications for extensibility as well as

[2] We adopt the Prolog convention to use upper-case letters for variables and _ for anonymous variables.

separate checking and will be discussed later on in detail (Sec. 6.3.2).

2.4 Lack of Modularity

Logic metaprogramming – like most other metaprogramming constructs – does not provide any modular checks making sure that computed programs or their clients are well-formed. If we look at the metaprogram and at `f3` in Fig. 1, then it is obvious that neither of them can be checked in a modular way, using only the interfaces of other referenced entities.

The code for a composed function in Fig. 1 is only available after instantiation with two concrete functions, and it is not obvious how to make sure that the program generator will only generate well-formed functions. On the other hand, the function generator does not have an interface that could hide its implementation details, hence the call to `compose(f1,f2)` in `f3` can only be type-checked by considering the full implementation (and computing the specific instance) of the function generator.

Consequently, errors are only detected after the generated programs are rejected by the object language compiler or interpreter. At this point, however, it becomes very difficult to understand the origin of the error, since the error occurs in generated code whose form depends on possibly complex computations. It is obviously desirable to check metaprograms and their clients once and for all in a modular way instead.

3. Modular LMP

The main reason why separate checking is not possible in LMP is that there is no interface mechanism which can describe the behavior of program generators in a way that enables information hiding, yet contains sufficient information to check each module separately. In the following, we will hence revisit our example and discuss what kind of information must be available in the interface and what properties the checking algorithm must have such that separate checking is possible. After this informal discussion, we will generalize our findings in the form of a formal definition in Sec. 4, which can also be read in parallel with this section. We will also identify various requirements on the logic solver that do not hold for standard Prolog-like solvers, but we will defer the discussion of how we addressed these requirements until Sec. 5.

A variant of Fig. 1 as a set of modules is shown in Fig. 2. Each module starts with the keyword MODULE and consists of up to four sections: REQUIRES, IMPLEMENTATION, PROVIDES, and IMPLEMENTATIONPROVIDES. The REQUIRES part describes the constraints on the context under which the module can be used. The IMPLEMENTATION part gives the implementation, and the PROVIDES part states what is exported to the external world. The IMPLEMENTATIONPROVIDES section is just a shorthand for copying definitions into both the IMPLEMENTATION and the PROVIDES part.

```
MODULE      % We call this module m1 in the text
IMPLEMENTATION
  funimpl(f1, add(5, arg)).
PROVIDES
  fun(f1).

MODULE      % We call this module m2
REQUIRES
  fun(f1).
IMPLEMENTATION
  funimpl(f2, add(7, call(f1, arg))).
PROVIDES
  fun(f2).

MODULE    % We call this module m3
REQUIRES
  fun(compose(f1,f2)).
IMPLEMENTATION
  funimpl(f3, add(arg,
                 call(compose(f1, f2), 13))).
PROVIDES
  fun(f3).

MODULE    % We call this module compose
IMPLEMENTATION
  funimpl(compose(A, B),
          add(call(A, arg), call(B, arg)))
    ← fun(A), fun(B).
PROVIDES
  fun(compose(A,B)) ← fun(A), fun(B).

MODULE    % We call this the system module
IMPLEMENTATIONPROVIDES
  fun(F) ← funimpl(P, Body),bodyok(Body).
  bodyok(arg).
  bodyok(N) ← number(N).  % number is a built−in
  bodyok(add(E1, E2)) ← bodyok(E1), bodyok(E2).
  bodyok(call(F, A)) ← fun(F), bodyok(A).
```

Figure 2. Modular version of Fig. 1

In our module system, every imported feature of another module must be described explicitly in the module's REQUIRES part. This enables completely separate checking; no other module is "referenced" (our modules do not have names anyway); rather, required and provided features are matched during module composition.

While the enumeration of every feature expected of another module may seem a bit cumbersome, it is the only way in which a module can be checked completely independent of its context. However, there could be different ways to arrive at this specification. For example, the requirements could be inferred from the implementation by an appropriate algorithm, or a lightweight device (such as a preprocessor) that allows one to import everything that is exported by another module (such as header files in C or imports in Java) could be implemented. Which (if any) of these mechanisms is used is not relevant for this work. Our modules could be viewed as a kind of intermediate layer, where conventional import statements are transformed into the required form by copying the referenced module's PROVIDES to the module's REQUIRES part. However, for this work we just assume that all requirements are explicitly specified.

The code in Fig. 2 contains one module that was not present in Fig. 1: the *system module*. The system module is special in that its PROVIDES part is implicitly part of the REQUIRES part of each other module; apart from this special role it is just another module. It is hence similar to, say, the `java.lang` package in Java or the "Prelude" in Haskell. In our system, however, we can use this module to define the static semantics and well-formedness of our language. We can see that the system module in Fig. 2 defines a simple type checker for FP[3].

3.1 Validity of Modules

Let us now discuss the meaning of the different parts of our modules. In traditional module systems, the PROVIDES and REQUIRES parts of a module consists of a list of names or type signatures, and a module is *valid* if, under the assumption that if all *required* identifiers are bound to implementations of the correct type, the implementation part of the module gives correct definitions of all identifiers/types in the PROVIDES part. Furthermore, two such modules are compatible, if they do not require an identifier to be bound to contradicting types and they do not export the same name [8].

In the case of our modules, a module is *valid* if each formula in the PROVIDES part is a *logical consequence* of the declarations in the REQUIRES part and the declarations from the implementation. In the example, the modules provide formulas of the form `fun(F)`. It is easy to see that both module `m1` and module `m2` are valid, and that validity coincides with type safety as defined by the system module. Note that both modules can be checked completely separately, using only the PROVIDES part of the system module as external input.

Module `m3` can now also be checked separately, due to the explicit requirement `fun(compose(f1, f2))` in its REQUIRES part. In particular, it can be checked independently of the metaprogram for composing the functions. This solves the first problem identified in the previous section: The lack of separate checking of clients of the code generator.

The more interesting challenge is the well-formedness check of the `compose` function generator, because it involves proving formulas containing universal quantification (variables are implicitly universally quantified) and implication. To this end, let us analyze the derivation of the `fun(compose(A, B))` $\leftarrow$ ... property at the end of the `compose` module. The property is universally quantified over `A` and `B`. In proof theory (and in our solver), the deduction rule for universal quantification is $\frac{\Gamma \vdash P(c)}{\Gamma \vdash \forall X.P(X)}$, where c is a fresh constant, hence let `a` and `b` be fresh constants for these variables. This leaves us with the task of proving

$$R \cup I \vdash \texttt{fun(compose(a,b))} \leftarrow \texttt{fun(a), fun(b)},$$

where R and I are the formulas from the requirement and implementation part of the module, respectively. The deduction rule for implication is $\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$, hence we must prove `fun(compose(a, b))` in the context $R \cup I \cup \{$`fun(a)`, `fun(b)`$\}$. Now type-checking the function calls in the function generator will succeed, because when the type-checker queries `fun(a)` and `fun(b)`, respectively, in the last clause of the `bodyok` predicate, this query succeeds because they are in the context. Hence the interface of the function generator is now also sufficient to check, once and for all, that all functions generated by this computation will be well-formed - which solves the second problem identified in the previous section: The lack of separate checking of code generators/meta programs.

3.2 Consistency and Compatibility

So far, we have only looked at the problem of checking a single module separately. However, the purpose of a module is to be composed with other modules, which raises the issue of module compatibility. A trivial solution would be to regard all modules as mutually compatible, but this would mean that many common compatibility requirements in real languages could not be modeled within our system, like forbidding two modules to export the same name or to create a cycle in an inheritance hierarchy.

Such constraints arise frequently in conjunction with metaprogramming. For example, a typical pattern for variability management in C++ is to use `#ifdef` preprocessor directives to switch between different variants of a class or method. Some configurations of the switches of `#ifdef` directives are often contradictory because, say, the same method would be defined twice. To enable separate checking, it is hence important that one can state these dependencies in the module's interface.

In our setting, we model incompatibility between modules as interfaces that are contradictory – that is, the union of the interfaces of the modules is inconsistent. Inconsistency cannot be directly expressed using Prolog, because sets of Horn clauses are always consistent by construction. A common way [31] to deal with this problem in a resolution-compatible way is to introduce a distinguished constant $\perp$ to act as a symbol for inconsistency. This constant can be used in heads of clauses and a proof of $\perp$ from a given set of formulas marks this set as inconsistent.

For example, a module may define two variants of a function `f1`, one of which calls a conditionally included function

[3] For simplicity we have shown a type checker that cannot deal with recursive functions. One way to support recursive functions is to exchange the goal `bodyok(Body)` in the definition of `fun(F)` by `bodyok(Body)` $\leftarrow$ `fun(F)`. This is not standard Prolog code but supported by our solver.

f2. The switches for these functions are a, b, and c. The dependencies can be expressed as follows:

```
MODULE
REQUIRES
  ⊥ ← a,b.
  c ← b.
IMPLEMENTATION
  funimpl(f1, add(5, arg)) ← a.
  funimpl(f1, add(7, call(f2,arg))) ← b.
  funimpl(f2, add(9, arg)) ← c.
PROVIDES
  fun(f1)← a.
  fun(f1)← b.
  fun(f2)← c.
```

The check for module validity will take care that this module is well-formed for every combination of the switches that satisfy the requirements (notice that the call to f2 in the second variant of f1 will only type-check due to the c ← b constraint in the REQUIRES part). Module compatibility makes sure that this module will never be composed with a module or a set of modules that make it possible to derive both the a and the b switch; in this case, the system would be able to prove $\perp$ and would mark this set of modules as incompatible.

With this notion of inconsistency, we can now define what it means for modules to be consistent and compatible with other modules. First, a module is *consistent* if it is valid, and if its interface (i. e., REQUIRES ∪ PROVIDES) is consistent as a logic theory, that is, it is impossible to deduce $\perp$. Furthermore, a set of modules is *compatible*, if the union of their interfaces is consistent. It is important that module compatibility depends only on the interface but not the implementation because module compatibility should not depend on implementation details. Later we will formulate a theorem that module composition cannot "go wrong" if only consistent and compatible modules are composed, and that module compatibility is preserved by module composition.

A weaker notion of compatibility checking of a set of modules would be to check compatibility only pairwise. However, this would not be sound. For example, module 1 may provide a, module 2 may provide b, and module 3 may require $\perp \leftarrow$ a, b. These modules are pairwise compatible, yet their composition would lead to an inconsistent module.

3.3 Information Hiding

Information hiding in our system is formalized by the requirement that the union of requirements and implementation must logically *imply* (but not be logically equivalent to) the PROVIDES part. However, since sets of logic (implementation) formulas may be inconsistent, we have to be careful that the implementations of two modules are not incompatible (meaning: their union is inconsistent) although their interfaces are compatible. For example, if we would not restrict the kinds of formulas in the implementation part, the following module would be consistent and compatible with every other module (since its interface is empty), yet its implementation is incompatible with any module that provides a function foo:

```
MODULE
IMPLEMENTATION
⊥ ← funimpl(foo, arg).
```

This example illustrates that we need more fine-grained control over the kind of information that can be hidden in module implementations. It must not be possible to put formulas into the implementation that may "silently" conflict with implementations and interfaces of other modules. Just forbidding the use of $\perp$ in the implementation part does not solve the problem, since one module may hide a function foo in its implementation part, and another module may contain a ⊥←funimpl(foo, arg) rule in its REQUIRES part.

To deal with this problem, we make a distinction between formulas that can be used to describe hidden implementation details (*implementation formulas*) and formulas that are used to describe module interfaces. Implementation formulas have the property that they are always consistent with arbitrary sets of other implementation formulas, and thus can be used in the implementation without further restrictions. If we want to use interface formulas in the implementation, there would be the possibility of an inconsistency. To prevent a module from hiding this possible source of inconsistencies, we require that all interface formulas that appear in the implementation must also appear in its PROVIDES part. The details of this mechanism will be discussed in the formal model presented in the next section.

The choice of implementation formulas and how the aforementioned properties of implementation formulas can be ensured depends on the logic. In our sample logic, we denote this distinction by a separation of the term constructors into two classes: the name of implementation constructors ends with impl (such as funimpl), and interface constructors are all other constructors. We then define a formula to be an implementation formula, if it uses an implementation constructor in its head. The consistency check for a module will then determine whether the module is consistent under any possible assignment of truth values to implementation literals - more about that later.

3.4 Composition, Finalization, and Code Generation

Consistent modules that are compatible can be composed to form a new module, whose PROVIDES and implementation parts contain the union of the PROVIDES and implementation parts of the constituents, respectively. The REQUIRES part contains the union of the REQUIRES parts of the constituents except those formulas that follow logically from the PROVIDES parts of the constituents. In the formalization, we will see that the computation of the new REQUIRES interface is actually a bit more complicated due to the fact that there may be circular dependencies and that the minimal set of requirements that leave the module in a consistent state is not unique. However, we later will show that the composition of two valid, consistent and compatible modules is again consistent and valid, and that the final result of composing a set of modules, which together constitute a complete program, is unique again.

At some point, all modules of the system have been composed and it is time for the backend to generate source code. We could define a (composed) module to be complete if its REQUIRES part is empty. However, this would be too strict, because a module may have formulas in the REQUIRES part that are not provable, yet are true if we consider the program to be complete. For example, the constraint $\perp \leftarrow$ `a,b` from above cannot be proven in a monotonic logic: While it may hold in some configuration of modules that not both `a` and `b` are true, the addition of a new module could always invalidate that property.

The source of this problem is that we expect our deduction relation to be *monotonic*, which means that whenever we can prove f from a set F, then we can also prove f from any set $F' \supset F$. Some well-formedness constraints such as the one described above are inherently closed-world, however. Other typical closed-world constraints use universal quantification, whereby the quantification is supposed to quantify over all entities in a system, hence the proof-theoretic rule for universal quantification is not applicable.

To deal with such closed-world constraints, it is necessary to mark the point when a program is complete, i.e., no more modules will be added to the program. When a program is sealed in this way, it is safe to switch to an extended closed-world version of the prover, which is no longer required to be monotonic. To this end, we introduce a *finalization* operation on a (composed) module, which attempts to prove all open requirements but this time using a *closed-world* version of the inference relation, where, say, $\perp \leftarrow a, b$ is considered proved if not both a and b can be deduced.

If finalization is successful (all requirements have been proved), the interfaces of the modules are discarded and the backend can generate code from the implementation of the finalized module. In modular logic metaprogramming, code generation always operates on the set of all implementation formulas. There is no separate compilation of modules, because the structure of the code that would result from a module's implementation may depend on the implementation of other modules and thus we cannot generate any meaningful code before we seal the composition of all modules.

3.5 Soundness

We have seen how various kinds of well-formedness conditions can be modeled in our modular LMP approach. However, our approach itself is not specific to any object language or set of well-formedness conditions, and hence our system can only detect well-formedness violations that have been modeled correctly. Therefore we can only guarantee relative soundness, that is, soundness with respect to the part of the static semantics that has been modeled, e.g., in the form of well-formed rules in the system module.

A point which makes the correct formulation of well-formedness constraints subtle is its interplay with the notion of information hiding embodied by our module system. A module can contain utter garbage in its implementation part if nothing is promised in its PROVIDES part. If a module contains, say, a function definition which is not exported and not called (directly or indirectly) by any other function which is exported, then a type error in this function cannot be detected. However, the well-formedness constraints can be specified in such a way that the type-check of an exported function requires all transitively called functions within the module (including those that are not exported) to be well-typed, too. Our type checker in Fig. 2 has this property.

For this reason it is important that the code generator will generate object language code only for those parts of the database that are used in the the proof trees required to prove the PROVIDES part of the module.

We believe that this is sufficient to model the well-formedness constraints of object languages in such a way that all well-formedness errors are detected within the module system – but only if the logic is expressive enough, e.g., computationally complete.

For simple logics, such as propositional logic or variants thereof, complex type-checking rules cannot be modeled within the logic. This does not mean, however, that our framework cannot be used. The key idea is to model those program entities which are too complicated (or undesired) to check within the logic as atomic constants. Then these entities can be checked by an external tool *before* the MLMP checking process takes place. Due to their representations as constants, the entities cannot be manipulated within the logic and will hence still be well-formed when they are later again reified as object-language code in the backend.

In general, the level of detail in the program representation hence has to fit to the expressive power of the logic and the complexity of the well-formedness constraints at that level of detail.

4. Formalization

In this section, we generalize from the discussion in the previous sections and present an axiomatization of a class of logics together with a formal definition of modules on top of the logic under which separate checking will be sound. We call a logic that fits to the axioms a *module description logic* (MDL).

The logic $\mathcal{F}_{P+}$, which was informally used in the previous section, is just one particular MDL, and we will give a more detailed description of $\mathcal{F}_{P+}$ and other possible logic languages later. The proofs for all theorems presented in this section are contained in the appendix.

4.1 Module Description Logic

We assume that the formulas in an MDL are given by a set $\mathcal{F}$. In the examples in the previous section, this set of formulas was the set of $\mathcal{F}_{P+}$ facts and rules. In the following, we use upper-case letters F, G to denote subsets of $\mathcal{F}$, and lower-case letters f, g to denote single formulas.

We also assume a deduction relation, $\vdash \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$, which defines what logically follows from a given set of formulas. We use the notation $F \vdash F'$ to denote that for all $f \in F'$ we have $F \vdash f$. The choice of this deduction relation is not completely arbitrary, because we need to have some important properties in the logic language to make the module system well-defined, in particular to prove anything about the result of combining modules.

The following definition sums up the properties of an MDL.

DEFINITION 1 (Module Description Logic). *A* module description logic *is a tuple* $(\mathcal{F}, \hat{\mathcal{F}}, \vdash, \vdash^{CW}, \bot_{\mathcal{F}})$ *satisfying the following conditions:*

$$\bot_{\mathcal{F}} \in \mathcal{F} \tag{1.1}$$

$$\text{If } F \vdash F' \text{ and } (F \cup F') \vdash F'' \text{ then } F \vdash F'' \tag{1.2}$$

$$\text{If } F \subseteq F' \text{ and } F \vdash G \text{ then } F' \vdash G \tag{1.3}$$

$$\text{For all } F \subseteq \mathcal{F} \text{ we have } F \vdash F \tag{1.4}$$

$$\hat{\mathcal{F}} \subseteq \mathcal{F} \tag{1.5}$$

$$\hat{\mathcal{F}} \nvdash \bot \tag{1.6}$$

$$\text{If } F \vdash f \text{ then } F \vdash^{CW} f \tag{1.7}$$

The first requirement (1.1) defines the notion of inconsistency. An important point we have stressed in the last section was the possibility and identification of inconsistencies between module interfaces. To make minimal assumptions about the structure of formulas, we only require that $\bot$, standing for inconsistency, is a logic formula.

The next three requirements are that $\vdash$ is *transitive* (1.2), *monotonic* (1.3) and *reflexive* (1.4). Transitivity means that if a set F' of formulas follows from another set F by deduction, and F'' follows from F', then F'' already follows from F. This is important to make sure that what follows from an interface also follows from an implementation of the interface. Monotonicity means that if a formula can be proved from a set F of formulas, then it can also be proved from every larger set that contains F. This property is essential for modularity, because it guarantees that requirements that are checked against interfaces of one module also hold when checked against a larger interface, for example the interface of a composition of modules. Reflexivity means that everything that is in the context can be deduced.

As we argued in the previous section, we have to distinguish a subset of formulas that is always consistent, which we call *implementation formulas*. We denote this set of formulas $\hat{\mathcal{F}}$ (1.5, 1.6). Finally, we need a closed-world version $\vdash^{\mathrm{CW}}$ of the deduction relation which must be an extension of the ordinary deduction relation that is no longer required to be monotonic (1.7).

4.2 Modules, Validity, and Consistency

Based on an arbitrary but fixed MDL $(\mathcal{F}, \hat{\mathcal{F}}, \vdash, \vdash^{\mathrm{CW}}, \bot_{\mathcal{F}})$ we can now define what a module is:

DEFINITION 2 (Module). *A* module *is a triple* (R, I, P) *of finite subsets of* $\mathcal{F}$.

In correspondence with the previous section, we call R the REQUIRES *part*, I the *implementation part* (the formulas between BEGIN and END), and P the PROVIDES *part*. We call the set $R \cup P$ the *interface* of the module. We do not model the IMPLEMENTATIONPROVIDES part that was used in the motivation, since it is just syntactic sugar for putting copies of formulas in both I and P.

Not every combination of requirements, implementation, and provided interface forms a *valid module*. To formalize the requirements for valid and consistent modules, we generalize the requirements formulated in Sec. 3.1 and 3.2. First we define what it means that a module is *valid*:

DEFINITION 3 (Valid Module). *A module* $M = (R, I, P)$ *is* valid, *if* $R \cup I \vdash P$.

Valid modules have correct specifications in the sense that their PROVIDES part is really a logical consequence of the implementation and the requirements. In other words, the system can trust the promises in the interface of the module without referring to its implementation. However, a module may still create inconsistencies, either by inconsistent formulas in the interface or by containing interface formulas in the implementation. For example, a module may require $\bot$, or hide $\bot \leftarrow x$ in its implementation.

We deal with these problems by defining a stronger property of modules, namely consistency. To do so, we first formalize the idea of a legal interface, which represents interfaces that are consistent and will never create inconsistencies with hidden parts of other modules:

DEFINITION 4 (Legal interfaces). *For a set* $F \subseteq \mathcal{F}$ *of formulas, we say that* F *is a* legal interface*, written ifc*(F)*, iff* $\forall F' \subseteq \hat{\mathcal{F}}.\quad F \cup F' \nvdash \bot$.

A valid module is consistent if it has a legal interface and all implementation formulas are either part of the information hiding space $\hat{\mathcal{F}}$ or announced in the PROVIDES part.

DEFINITION 5 (Consistent Module). *A module* M *given by the three sets* (R, I, P) *is* consistent*, written module-ok*(M)*, iff*

$$M \text{ is valid} \tag{5.8}$$

$$\mathit{ifc}(R \cup P) \tag{5.9}$$

$$\forall f \in I.(f \in \hat{\mathcal{F}}) \vee (f \in P). \tag{5.10}$$

By 5.10, consistency guarantees that we can reason about the possible effects or inconsistencies caused by a module by looking at its interface only. Together with validity, consistency defines the separate check that can be made on each module in isolation. Any further checks will only concern the interface of the modules. Hence, module-ok(M) can be seen as our formalization of the requirements that enable separate checking.

4.3 Composing Modules

The composition of two or more modules requires two activities: A check whether the modules are compatible, and a process which discharges those requirements of the modules that are exported by the other module and merges the definitions of the modules. Compatibility is defined as follows:

DEFINITION 6 (Module compatibility). *A set of modules* $\mathbb{M} = \{(R_1, I_1, P_1), \ldots, (R_n, I_n, P_n)\}$ *is compatible, written* $\div(\mathbb{M})$*, iff*

$$\mathit{ifc}(\bigcup_{i\in\{1,\ldots,n\}} R_i \cup P_i).$$

If only two modules are involved, we write $M \div M'$ *to stand for* $\div(\{M, M'\})$*.*

It is crucial that the compatibility between modules does not depend on their respective implementation. Hence it would in principle be possible that the implementations are contradictory. But if all modules are consistent, then the fact that these modules are compatible also means that their implementations are compatible:

THEOREM 7. *If a set* $\{(R_1, I_1, P_1) \ldots, (R_n, I_n, P_n)\}$ *of modules is compatible and every module* M_i *is consistent, then* $\cup_{i\in\{1,\ldots,n\}} I_i \not\vdash \bot$*.*

As a consequence, we can merge these implementations without the danger of inconsistencies, and we can then create a new module for the merged implementation by creating a new interface. This interface contains only those formulas of the original modules' interfaces that are not proved by the combined provided parts. Module reduction is a formalization of this process as a transition system on modules:

DEFINITION 8 (Module Reduction). *A module* (R, I, P) *is* reducible by $f \in R$*, if* $P \vdash f$ *and* $(R \setminus \{f\}, I, P)$ *is valid. The reduction relation on modules is defined as*

$$(R, I, P) \leadsto (R \setminus \{f\}, I, P) \text{ if } (R, I, P) \text{ is reducible by } f.$$

The reflexive and transitive closure of $\leadsto$ *is* $\leadsto^*$*.* M' *is a* minimal reduction *of a module* M*, written* $M \leadsto_{\mathit{min}} M'$*, if* $M \leadsto^* M'$ *and* $(\neg\exists M'')\ M' \leadsto M''$ *(*M' *is a normal form of* M*).*

The validity check of $(R\backslash\{f\}, I, P)$ can fail in the case of non-wellfounded circular definitions. For example, a module may require `fun(f)` and provide `fun(f)`. Such situations can, in the general case, not be detected in a modular way, which explains why we have to look into the implementation part. Rather than throwing an error, we have decided to just not allow the reduction of this requirement, since the situation may still be resolved by adding another module later on which adds the missing feature.

Module reduction preserves consistency.

LEMMA 9 (Preservation of consistency under reduction). *If module-ok*(M) *and* $M \leadsto M'$*, then module-ok*(M')*.*

The composition of two modules $M_1 = (R_1, I_1, P_1)$ and $M_2 = (R_2, I_2, P_2)$ is defined as the set of all possible minimal reductions of composing the parts of both modules:

DEFINITION 10 (Module composition).

$$M_1 + M_2 = \{M \mid (R_1 \cup R_2, I_1 \cup I_2, P_1 \cup P_2) \leadsto_{\mathit{min}} M\}.$$

This definition of module reduction will correctly discharge circular dependencies between modules. For example, if

$M_1 = (\{\texttt{fun(a)}\}, \{\texttt{funimpl(b, call(a,arg))}\}, \{\texttt{fun(b)}\})$

and

$M_2 = (\{\texttt{fun(b)}\}, \{\texttt{funimpl(a, call(b,arg))}\}, \{\texttt{fun(a)}\})$

and $M \in M_1 + M_2$ then

$M \leadsto_{\mathrm{min}}$
$(\emptyset, \{\texttt{funimpl(a,...),funimpl(b,...)}\}, \{\texttt{fun(a),fun(b)}\})$

Module composition produces a unique new module, unless the module contains non-wellfounded circular dependencies. For example, if $M_1 = (\{a\}, \{b \leftarrow a\}, \{b\})$ and $M_2 = (\{b\}, \{a \leftarrow b\}, \{a\})$ are combined, the resulting requirements can be reduced to $\{a\}$ or $\{b\}$, but no further, since the module would become invalid otherwise. However, we will later show that *all* of the modules in $M_1 + M_2$ are equivalent with regard to the final program at the end of the composition process.

DEFINITION 11 ($\vdash$-Equivalence). *We say that two modules* $M_1 = (R_1, I_1, P_1)$ *and* $M_2 = (R_2, I_2, P_2)$ *are* $\vdash$-equivalent*, written* $M_1 \approx_\vdash M_2$*, if*

$$I_1 = I_2 \text{ and } P_1 = P_2 = P,$$
$$P \cup R_1 \vdash R_2 \text{ and } P \cup R_2 \vdash R_1$$

THEOREM 12 (Composition and $\vdash$-Equivalence). *If both* M' *and* M'' *are minimal reductions, i.e.,* $M', M'' \in M_1 + M_2$*, then* $M' \approx_\vdash M''$*.*

We can pick any of the minimal reductions without losing information and thus can choose an arbitrary deterministic module composition.

DEFINITION 13 (Deterministic Composition). *We define the* deterministic composition *of two modules* M_1 *and* M_2 *as*

$$M_1 \oplus_\sigma M_2 = \sigma(M_1 + M_2),$$

for some selection function $\sigma : S \mapsto m \in S$*.*

The $\oplus_\sigma$ composition operation has the property that it preserves consistency and validity of modules:

THEOREM 14 (Composition). *Let M_1 and M_2 be two modules with module-ok(M_1) and module-ok(M_2) and $M_1 \div M_2$. Then module-ok$(M_1 \oplus_\sigma M_2)$.*

Let us now come to the composition of sets of arbitrary many modules. For this we must first establish that composition preserves compatibility:

THEOREM 15 (Preservation of Compatibility).

$$\text{If } \div(\{M_1, M_2, M_3\}), \text{ then } (M_1 \oplus_\sigma M_2) \div M_3$$

The order in which modules are composed is not significant:

THEOREM 16. *The operation $\oplus_\sigma$ is associative with respect to $\vdash$-equivalence:*

$$(M_1 \oplus_\sigma M_2) \oplus_\sigma M_3 \approx_\vdash M_1 \oplus_\sigma (M_2 \oplus_\sigma M_3).$$

Thus we can write $\bigoplus^\sigma_{1\leq i\leq n} M_i$ for the result of iteratively combining adjacent modules from the list $(M_i)_{1\leq i\leq n}$ in arbitrary order.

This means that we can generalize Thm. 14 to sets:

THEOREM 17 (Composition of module sets). *Consider a set $\mathbb{M} = \{M_1, \ldots, M_n\}$ of modules with module-ok(M_i) for all i and $\div(\mathbb{M})$. Then module-ok$(\bigoplus^\sigma_{1\leq i\leq n} M_i)$.*

4.4 Finalization and Code Generation

Using the composition operator defined in the previous paragraph on all modules of a program, we obtain a single module representing our program, but this resulting module may still contain formulas in its REQUIRES part even if all dependencies are satisfied. As discussed in Sec. 3.4, these formulas are closed-world requirements that cannot be proved with the monotonic deduction system.

To generate code for a program represented by a module we have to ensure that the module's REQUIRES part is empty, because otherwise required information could be missing in the final program or there could be unsatisfied constraints. As discussed before, this cannot be achieved using $\oplus$, because composition uses the $\vdash$ deduction relation, which is monotonic by definition. Monotonicity is necessary for composition, because the resulting module may be composed with other modules later, but as soon as we want to generate code we know that there will be no more compositions. What we need is an operator that turns the final module into a program or indicates an error, if one of the closed-world requirements is not satisfied. We call this operation *finalization*.

For finalization we use the (typically) non-monotonic extension $\vdash^{\text{CW}}$ of the deduction relation $\vdash$. There are different ways to define non-monotonic extensions such as negation-as-failure [36], and the choice of the non-monotonic extensions determines which kinds of non-monotonic well-formedness requirements can be declared. Usually, the closed-world extension will contain additional deduction rules such as $\frac{F \nvdash^{\text{CW}} f}{F \cup \{f\} \vdash^{\text{CW}} \bot}$. For the purpose of our formalism, the exact kinds of non-monotonic extensions are not significant; the only role of $\vdash^{\text{CW}}$ is to discharge nonmonotonic requirements, hence it only matters that the deduction relation is an extension of the monotonic one that is no longer required to be monotonic, as required by Def. 1.7.

The result of finalization is what we call a *program*, namely the database of all implementation formulas which forms the input to the code generator backend. If finalization succeeds, neither the REQUIRES nor the PROVIDES part are needed anymore. For the definition of finalization we use $\rightsquigarrow^{\text{CW}}$, a variant of $\rightsquigarrow$ that is defined by using $\vdash^{\text{CW}}$ instead of $\vdash$.

DEFINITION 18 (Closed-World Module Reduction). *A module (R, I, P) is* closed-world reducible *by $f \in R$, if $P \vdash^{CW} f$ and $(R \setminus \{f\}, I, P)$ is valid. The reduction relation on modules is defined as*

$$(R, I, P) \rightsquigarrow^{CW} (R \setminus \{f\}, I, P)$$

if (R, I, P) is closed-world reducible by f. The reflexive and transitive closure and minimal reductions of $\rightsquigarrow^{CW}$ are defined like in Def. 8.

It is obvious that $\rightsquigarrow^{\text{CW}}$ preserves validity and consistency. With this definition, finalization is defined as follows:

DEFINITION 19 (Finalization).

$$\mathit{Final}(R, I, P) = \begin{cases} I & \text{if } (R, I, P) \rightsquigarrow^{CW}_{min} (\emptyset, I, P) \\ \mathit{failure} & \text{otherwise} \end{cases}$$

The following theorem sums up the main properties of our approach. By this theorem we know that in whatever order we compose our modules, the result is well-defined if the comprising modules are.

THEOREM 20. *Given a set $\{M_1, \ldots, M_n\}$ of consistent and compatible modules, a selection strategy σ, and the composed module $M = (R, I, P) = \bigoplus^\sigma_{1\leq i\leq n} M_i$, the following statements hold:*

- *M can be computed in a finite number of reduction steps.*
- *M is consistent, i.e., module-ok(M).*
- *If σ and σ' are two selection functions (Def. 13) and $\oplus^\sigma$ and $\oplus^{\sigma'}$ are their associated composition operators, then Final$(\bigoplus^\sigma_{1\leq i\leq n} M_i) =$ Final$(\bigoplus^{\sigma'}_{1\leq i\leq n} M_i)$.*
- *The implementation part of M is consistent, i.e., $I \nvdash \bot$.*
- *If Final$(M) = I \subseteq \mathcal{F}$ then $I \vdash P$.*

5. Implementation

We have implemented a system for MLMP which is built according to the formal model described in the last section. As in the formal model, our MLMP implementation can be parameterized with an MDL and a corresponding solver, and is independent of any particular object language. We have also developed a prototypical system module for the Java programming language. In the following, we describe both the MLMP system and our solver for the $\mathcal{F}_{P+}$ module description logic.

The system and the solver are available for download at `http://daimi.au.dk/~klose/mlmp`.

5.1 The MLMP System

We have implemented our MLMP system in SWI-Prolog[4]. It comprises a central module, the module composer, which is defined in terms of the three variation points of the system: the frontend interface, the MDL interface with corresponding system module, and the backend interface.

The *frontend*, which is responsible for reading module descriptions, can support multiple input formats. There is a "raw" input format, namely input terms of the form $\texttt{mod}(R, I, P)$ which describe the three parts of a module. To ease writing modules, the frontend can be equipped with functions to accept quoted object language code and transform it into formulas accepted by the system, or to convert files written directly in the object language.

The *module composer* is the central component of our implementation, which provides the top level interface to the user and coordinates the use of the frontend, the MDL implementation, the system module, and the backend. The module composer takes a list of modules as input, checks each module for consistency using the MDL interface, and produces a composed module according to our formalization. If required by the user, the module composer will also try to finalize the module and, if successful, produce code using the backend. We have developed a sample code generator component that produces Java code from $\mathcal{F}_{P+}$ modules.

Whenever module checks or reduction are necessary, the system invokes the deduction relation through the MDL interface. The system module is another parameter, and the module composer will take care of adding the system module constraints to all consistency and compatibility checks, as described in Sec. 3.1.

All in all, the system is a straightforward and direct implementation of our formalization.

5.2 The $\mathcal{F}_{P+}$ Module Description Logic

The logic we use in our examples, $\mathcal{F}_{P+}$, consists of the "pure" and monotonic core of Prolog [37] (which means: no negation-as-failure or other closed-world reasoning, no cuts, no side effects), and a few simple extensions well-known from λProlog [32, 33] and its extensions. Hence we do not claim any originality of our solver from the perspective of logic programming and theorem proving.

[4] `http://www.swi-prolog.org/`

We have implemented a solver for this logic in Prolog, as an extension of the standard meta-circular Prolog interpreter [37]. Although the Prolog extensions we need are available in λProlog, we have not used λProlog itself for two reasons: First, it is not clear whether the higher-order extensions of λProlog interfere with our definition of $\hat{\mathcal{F}}$ and the requirements of Def. 1. Rather, for this work we were looking for the simplest logic that fulfills our formal requirements and is sufficient for the examples we are interested in. Second, we need control over the deduction mechanism such that we can perform the consistency check (see next subsection) and switch to closed-world deduction for finalization, and this turned out to be much easier if we have our own solver implementation.

Programs in $\mathcal{F}_{P+}$ consist of standard Prolog Horn-clauses, the term $\bot$, which is used to denote inconsistency, and the following features from λProlog: implications as goals, explicit quantification, and universally quantified goals. The set $\mathcal{F}_{P+}$ of formulas in our language is defined over a set L of (positive) literals. There is a subset $\hat{L} \subset L$ which contains all implementation literals and for which $\bot \notin \hat{L}$. Syntactically, we use the notation from Sec. 3.3 and distinguish implementation literals from other literals by the functor suffix `impl`.

The set of implementation formulas is defined inductively as follows:

$$\hat{\mathcal{F}}_{P+} = \hat{L} \cup \{\hat{l} \leftarrow l_1 \wedge \ldots \wedge l_n, (\forall v)\hat{f},\ (\exists v)\hat{f}\}$$
$$\text{where } \ l_i \in L,\ \hat{l} \in \hat{L},\ \hat{f} \in \hat{\mathcal{F}}_{P+}.$$

Since $\bot$ cannot be used in the head and there is no negation, this definition satisfies (1.5) and (1.6) of Def. 1.

Formulas in $\hat{\mathcal{F}}_{P+}$ can be used both as clauses in the logic program and as goals, which makes it necessary to give an interpretation of implication and universal quantification. Formally, the set of $\hat{\mathcal{F}}_{P+}$ formulas is a subset of the *first-order hereditary Harrop formulas* (fohh) [32], and the proof method (which was sketched in the motivation in Sec. 3) for fohh agrees with the classical proof-theoretical semantics in predicate logic [32]. A detailed discussion of how a solver for such formulas works is not relevant for this paper. The basic structure of the solver is that of a standard resolution-based Prolog solver, and the additional features (quantification, implications as goals) are handled as described in [32].

For the interface formulas in $\mathcal{F}_{P+}$, $\bot$ is allowed as the head of a clause. Operationally we treat $\bot$ like a literal and we interpret the meaning of $\bot$ as *minimal-logic negation* [31, Sec. 7], that is, we are only interested in detecting inconsistencies but we do not allow deducting arbitrary formulas from a proof of $\bot$. The full set of formulas is given by the following inductive definition:

$$\mathcal{F}_{P+} = L \cup \{l \leftarrow l_1 \wedge \ldots \wedge l_n,\ (\forall v)f,\ (\exists v)f\}$$
$$\text{where } f \in \mathcal{F}_{P+},\ l \in L \cup \{\bot\},\ l_i \in L.$$

Regarding the remaining properties of Def. 1, (1.1) and (1.4) hold trivially, and (1.3) holds because we use only the monotonic core of Prolog and monotonic extensions. (1.2) holds modulo the depth bound of the solver, see discussion in the next section.

5.2.1 Checking Module Consistency

The definition of legal interfaces (Def. 4) involves checking the public interface of the module together for consistency with arbitrary sets of implementation formulas, and is hence instrumental to make information hiding sound. However, in $\mathcal{F}_{P+}$ it is not obvious how to implement this check, since it quantifies over an infinite set. Fortunately the solution is simple: During the consistency check, the solver assumes that every implementation literal is true. This check is sound because implications in $\mathcal{F}_{P+}$ can only contain positive literals. It is also complete, since there could always be an additional module which contains the respective implementation literal.

Formally we can describe this strategy by a mapping $T : \mathcal{F}_{P+} \rightarrow \mathcal{F}_{P+}$ which removes all implementation literals from a given formula. If the resulting body of the clause is empty, the head literal is returned and if the head literal is an implementation literal, it is replaced by true. This strategy for consistency checks is both sound and complete, as guaranteed by the following theorem:

THEOREM 21.

$$(\exists F \subseteq \hat{\mathcal{F}}_{P+})\ \{f_i\}_{i\in I} \cup F \vdash \bot \Leftrightarrow \{T(f_i)_{i\in I}\} \vdash \bot.$$

5.2.2 Closed-World Reasoning

As discussed in Sec. 4.4, the finalization phase of the module composition process requires a closed-world version of the deduction relation, which is no longer required to be monotonic. Our $\mathcal{F}_{P+}$ solver introduces the following two non-monotonic proof rules during finalization: First, a formula $\bot \leftarrow l_1 \wedge \ldots \wedge l_n$ is considered proved if any of the l_i cannot be proved (this is essentially negation-as-failure). Second, a universally quantified implication can be proved by iterating over all possible instantiations of the premise (in the SWI Prolog system this non-monotonic rule is available as the `forall` primitive).

5.2.3 Termination and Incompleteness of the Solver

Since $\mathcal{F}_{P+}$ is an extension of (pure) Prolog, the solver is incomplete, just like any Prolog solver. This means that the proof search is either not guaranteed to terminate, or the solver will sometimes answer *unknown* instead of *yes* or *no*. We have chosen the latter approach by limiting the search depth of the prover. If the limit is exceeded, the prover answers *unknown*. For our purposes, it is sufficient to make sure that the solver only errs on the "safe side". This means that whenever *unknown* occurs during a check (module validity, consistency, or compatibility), the check fails. The practical significance of incompleteness will be discussed in the next section.

```
MODULE
REQUIRES x: Nat
IMPLEMENTATION z : Nat = 3, f : Nat → Nat = λy.y+x+z
PROVIDES f : Nat → Nat
```

Figure 3. Module in **F1**

5.3 The Java binding

We have also implemented a prototypical system module and a code generator for the Java programming language. Obviously the static semantics of the Java programming language is much more complex than FP (our example from the introduction), but conceptually the system module for Java has the same shape as the one for FP in Fig. 2. We have choosen open representations for both classes and methods (cf. Sec. 2.3), which enables more flexibility from the perspective of metaprogramming - for example, methods can be added to a class 'externally'.

Our code generator just iterates over the database and computes, starting from a set of "start classes" transitively the code of all classes that are referenced from this set. A Java compiler is then used to compile the code to Java byte code.

6. Discussion

In this section we demonstrate the versatility of MLMP and discuss alternative MDLs, other applications, and limitations of the approach.

6.1 Alternative MDLs

In the previous section we introduced the MDL instance $\mathcal{F}_{P+}$ as a very expressive and undecidable language. However, $\mathcal{F}_{P+}$ is just one data point in a wide spectrum of possible MDLs, which also includes simpler, decidable logics. In the following we discuss other MDLs and their potential applications.

6.1.1 Simple Signature Matching

Our first example demonstrates how our module system is connected to traditional module systems that use simple lists of signatures in their export/import interfaces. For illustration, we choose Cardelli's module system for **F1** [8] as an example. An example for an **F1** module is given in Fig. 3.

The module system for this language has the following responsibilities: a) It must check that the modules are well-typed given the specified imports, b) it must check that what is promised in the PROVIDES part is available in the imple-

```
MODULE
REQUIRES bound(x,Nat)
IMPLEMENTATION
  def(l_1,Nat,3),
  def(l_2,Nat →Nat,λy.y + x + l_1),
  export(l_2,f)
PROVIDES
  export(l_2,f), bound(f,Nat→Nat)
```

Figure 4. Encoding of Fig. 3 in $F_{\mathbf{F1}}$

mentation, c) two modules must be deemed incompatible if they have conflicting expectations regarding the type of an imported term, d) two modules that export the same name must be deemed incompatible. Furthermore, the composition of two compatible modules should not produce accidental name clashes of hidden names, i.e., lexical scoping must be preserved.

In the following we present the Module Description Logic for **F1**. In particular, we show how the problem with the lack of name spaces and scoping can be solved in the context of our framework.

Lexical scoping is a subtle issue if modules are represented as flat sets of formulas. Module composition (Def. 10) combines modules by merging their implementation parts, and care must be taken to preserve scoping during this operation. We deal with this problem by a definition of $\hat{\mathcal{F}}_{\mathbf{F1}}$ which takes care that every name has a globally unique label l; technically, this could be realized by assigning, say, sufficiently large random numbers to every internal name. Formally, we define the set of formulas in the logic as follows (x, t, and e range over all names, types, and terms, respectively):

$$\hat{\mathcal{F}}_{\mathbf{F1}} = \{\mathtt{def}(l,t,e) \mid l \text{ is unique in } \hat{\mathcal{F}}_{\mathbf{F1}}\}$$
$$\mathcal{F}_{\mathbf{F1}} = \hat{\mathcal{F}}_{\mathbf{F1}} \cup \{\mathtt{export}(l,x), \mathtt{bound}(x,t)\} \cup \{\bot\}$$

The corresponding encoding of the example is given in Fig. 4. We represent the definition of names by def terms. Internal labels can be exported as name x via export. The bound predicate is used in the interface parts to specify imported and exported names. The deduction relation $\vdash$ over these formulas is given by the following four inference rules:

$$\frac{F \vdash \mathtt{def}(l,t,e) \quad F \vdash \mathtt{export}(l,x) \quad \mathit{welltyped}(F,l,t,e)}{F \vdash \mathtt{bound}(x,t)} \qquad \frac{f \in F}{F \vdash f}$$

$$\frac{F \vdash \mathtt{bound}(x,t) \quad F \vdash \mathtt{bound}(x,t') \quad t \neq t'}{F \vdash \bot} \qquad \frac{F \vdash \mathtt{export}(l,x) \quad F \vdash \mathtt{export}(l',x) \quad l \neq l'}{F \vdash \bot}$$

The first rule takes care of points a) and b) discussed above: If a module declares that it provides a definition of x with type t, then it will be defined and well-typed. We assume that a type checking algorithm is given in the form of a $\mathit{welltyped}$ predicate, which builds a typing context out of the F parameter and then checks that e has type t under this context. The $\mathit{welltyped}$ predicate must type-check exactly the term t and any other term in F that is (transitively) referenced by t – if it checks less, then hidden terms that are needed in the final program may not be well-typed. If it checks more, then the deduction rule would not be monotonic (1.3). The next rule is just set membership and guarantees (1.4) of Def. 1. In particular, this rule is responsible for matching signatures, i.e., required and provided bound formulas. The last two rules declare a set of formulas inconsistent if it has contradicting expectations with regard to the type of an identifier, or the same name is provided twice, respectively (see c) and d) above). The last rule also explains why the internal label l_2 is also published in the PROVIDES part of Fig. 4: It denotes that there is an actual hidden implementation associated to that label. Without this information, it would be impossible to detect whether another module has a definition of the same name. In fact, Def. (5.10) forces any consistent module to publish all its exports in its PROVIDES part.

It is not hard to see that this deduction system has the properties demanded by Def. 1. (1.1) and (1.5) are trivial, and we have already discussed (1.4). (1.2) is trivial since the logic has no implication or the like. (1.3) was already discussed for the first rule above, and the other rules are trivially monotonic. (1.6) is more interesting: If it were possible to deduce more bound formulas by adding more def formulas, then (1.6) would not hold due to the last two deduction rules. But due to the incorporation of the export formulas, which are not part of $\hat{\mathcal{F}}_{\mathbf{F1}}$, this cannot happen and (1.6) holds. Closed-world reasoning is not required for **F1**, hence $\vdash^{\text{CW}}=\vdash$, which trivially fulfills 1.7.

In a more expressive logic, such as $\mathcal{F}_{P+}$, $\mathcal{F}_{\mathbf{F1}}$ can be encoded *within* the logic, that is, the deduction rules given above are encoded as formulas within the system module of the logic. Hence the above encoding of the **F1** module system could also be performed in $\mathcal{F}_{P+}$.

6.1.2 Propositional Logic

Propositional logic also trivially fulfills Def. 1 by taking $\hat{\mathcal{F}} = \emptyset$. This logic is too simple to represent programs, but it can still be useful to express configuration well-formedness constraints, such as constraints on the valid configurations of a product line or feature model [4]. For example, in a feature model it is typical to have constraints of the form "feature X requires feature Y" or "feature X and feature Y cannot be used together", and such constraints can easily be formulated in propositional logic [4]. A nice property of our approach is that the compatibility and consistency checks are by construction again modular, as with every instance of our model. In contrast, configuration checkers for feature models are typically global checkers that need the complete configuration of a feature model in one place. In our model, this would correspond to checking all constraints not until finalization.

If necessary, the idea can easily be generalized to more powerful constraint systems for configuration checking, such as the one investigated by [5]. It is only necessary to make sure that Def. 1 holds.

6.1.3 Datalog

Datalog [9] is a subset of Prolog that is decidable and can be implemented very efficiently, as compared to full Prolog. These properties come at the price of restricted arithmetic, restricted negation, and the prohibition of complex terms as arguments. This restricted logic is not expressive enough to model complex constraints, such as a type systems, but it is sufficient to reason about simpler properties of software systems, for example source code querying, detection of design guideline violations, and enforcement of architectural constraints [18, 23]. We could combine these approaches with our module system to provide modular checks for architectural constraints on arbitrary collections of source code artifacts.

6.2 Other Applications

In the previous presentation of alternative MDLs we also discussed a set of applications of these logics. Since our $\mathcal{F}_{P+}$ logic can encode all these logics, these applications would work equally well in $\mathcal{F}_{P+}$. In this section, we want to discuss three more application areas that make full use of the expressiveness of $\mathcal{F}_{P+}$.

6.2.1 Design Pattern Generators

It is known that logic metaprogramming can be used to generate parts of the implementation of design patterns that depend on the structure of the rest of the system [15]. Typical examples are *visitors* for class hierarchies, *proxy/wrapper/decorator classes*, and *flyweight* objects [21]. The implementation of such design pattern consists of one or more class computations and a set of predicates through which the user can control the application of the class computations.

For example, a visitor generator for a class hierarchy with root X is realized by the module in Fig. 5, which uses the program representation from Sec. 3. This generator creates the visitor class `visitor(X)` and `visit` methods for all possible subtypes of X. It also takes care of creating all the necessary `accept` methods in the elements of the class hierarchy. The generation of a visitor can be triggered by another module by putting `make_visitor('MyClass')` into its PROVIDES interface (and the implementation). Just as in our function generator example, the code generator can be type-checked once and for all, and clients using visitors can be type-checked in terms of the visitor interface only.

```
MODULE
IMPLEMENTATION
 classimpl(visitor(X), 'Object') ←
    make_visitor(X), class(X).
 methodimpl(visitor(X), visit(Y), [y], [Y],
        void, return ) ←
    make_visitor(X),
    class(Y, S), subtypeeq(Y, X).
 methodimpl(Y, accept, [visitor], [visitor(X)],
        void, Body ) ←
    make_visitor(X),
    class(Y, S), subtypeeq(Y, X),
    Body = call(visitor, visit(Y), [this]).
PROVIDES
 class(visitor(X), 'Object') ←
   make_visitor(X), class(X).
 method(visitor(X), visit(Y), [Y], void) ←
    make_visitor(X), class(Y, _), subtypeeq(Y, X).
 method(Y, accept, [visitor(X)], void) ←
    make_visitor(X),
    class(Y, S), subtypeeq(Y, X).
```

Figure 5. Visitor generator

6.2.2 Cross-Language Typing

A particular strength of our approach is that it is independent of a specific object language, which makes it particularly simple to integrate different object languages. There are often complex constraints between different code artefacts written in different languages. For example, classes for database access may be generated from specifications in an XML format. One would like to typecheck clients of these classes against the XML specification rather than against generated code. Similarly, there are often complex well-formedness constraints between configuration files and ordinary source code. For example, a class name occuring in a configuration file for a component container may be required to be the name of an existing class which must be a subclass of a container class. Finally, it would also be desirable to have a fine-grained modular typing discipline for calls between program parts written in different languages. The lack of modular cross-language checks typically leads to subtle errors that occur late in the build process or sometimes not until deployment- or runtime.

All such cross-language well-formedness constraints can easily be modeled and hence checked in our framework. All a user of our framework has to do to integrate several languages is to provide a frontend for each of these languages, and then add the cross-language well-formedness constraints either to the system module or to an ordinary module that is imported by the modules to be checked.

6.2.3 Pluggable Type Systems

Another interesting application is to provide support for *pluggable type systems* [3, 6]. Pluggable type systems check optional or domain-specific well-formedness constraints that are not part of the type system of the object language. Typical examples include non-null type checkers or type systems for alias control.

In our setting, pluggable type systems can be defined in the PROVIDES part of a module, just like the normal base type system is defined in the system module, and other modules could then use the type system by importing it and exporting, say, `method_nonnull_ok` well-formedness certificates in their PROVIDES part, whereby `method_nonnull_ok` is defined by the type system module and holds if the respective method is well-formed with regard to the non-null type system. In fact, the type system discussed in Sec. 3 can already be considered a pluggable type system if the rules are moved from the system module into an ordinary module.

Another conceivable application of pluggable type systems would be to retroactively check an existing module according to a new type system (or, more generally, static analysis). However, there is one major problem with this idea: typically, the actual code to be type-checked is hidden in the *implements* part of the module, and hence another module that is not explicitly imported has no access to these hidden parts. The implementation details would have to be exposed in the PROVIDES part of a module to make this work, but this would be in conflict with information hiding. A possible solution to this problem would be a more fine-grained notion of information hiding, where, say, some modules are allowed to see more implementation details than other modules. The exact design of such a mechanism is part of our future work.

6.3 Limitations

We conclude the discussion with a consideration of the limitations of our approach.

6.3.1 Incompleteness of the Solver

Although we have seen that our formal framework is also applicable and useful if it is instantiated with simple, decidable logics, many interesting applications – in particular those related to type checking – require undecidable logics such as $\mathcal{F}_{P+}$. In Sec. 5.2.3 we have already discussed how incompleteness is being dealt with technically. Here we want to discuss the implications on the programming model.

The consequence of the incompleteness of a solver is that sometimes modules are unfairly rejected. The situations in which this can happen depend on the deduction algorithm of the solver. Our resolution-based $\mathcal{F}_{P+}$ solver, for example, is suspectible to the same programming patterns that lead to non-termination in Prolog programs, such as left-recursion. For example, if the transitive hull of the subtyping relation in our Java binding would be specified via a rule such as

```
subtypeeq(A,C) ← subtypeeq(A,B), subtypeeq(B,C).
subtypeeq(A,B) ← class(A,B).
```

then the solver will return *unknown* whenever this rule is used. Similarly, the backtracking algorithm may always backtrack to the wrong choice points, although another choice point would quickly lead to a successful answer. There are techniques to reduce the number of situations where the solver loops (such as tabling, or left-recursion elimination), but this does not change the fact that a user of our system has to encode his formulas in a way that is compatible with the solver algorithm, if the solver is incomplete, for example by rewriting the two rules to remove the left-recursion:

```
subtypeeq(A, A) ← class(A, S).
subtypeeq(A, C) ← class(A, B), subtypeeq(B, C).
```

Our $\mathcal{F}_{P+}$ implementation uses a depth-limit approach to avoid infinite loops in the prover, which means that programs of a certain complexity (i.e., requiring a proof of large depth) will be rejected even if they are correct. Our experience shows, however, that this is not a problem in practice, since most proofs have small depth.

6.3.2 Influence of the Representation

Under a closed-world assumption, such as in traditional LMP, the form of program representation is not very important, because different representations can easily be computed from each other. In our modular setting, however, the choice of representation directly influences which kinds of properties can be proven modularly, and which kinds of program computations can be expressed.

We have already hinted at the question of open versus closed representation in Sec. 2.3. In the program representation chosen in this paper, classes, methods, fields, and constructors were represented openly in the form of predicates, whereas we chose a closed representation of method- and constructor bodies as terms nested inside their respective method and constructor declaration.

Open representations are extensible: It is possible to add new classes/methods/fields/constructors without modifying any existing module code. Closed representations are not: It is not possible to add, say, another statement to a method body without changing the method body representation.

On the other hand, all properties that quantify over all entities of a kind (such as: all classes, all methods etc.) cannot be established modularly if these entities have an open representation. For example, with our open representation of methods it is not possible to check modularly that there is no ambiguous overloading of two methods. It is possible to specify such constraints, and violations against these constraints are detected "early" (as soon as two modules are composed that violate the property), but these constraints will stay in the REQUIRES part of the composed module and they will only be proved during finalization, when the solver switches to closed-world reasoning.

This is not a problem when entities have a closed representation. For instance, it can be established modularly, once and for all, that a method body is well-typed. Hence, the choice of the program representation is a tradeoff between extensibility and modularity.

In Sec. 6.1.1 we have seen that dealing with names, name clashes, and lexical scoping properly also requires a carefully chosen program representation. The fact that formulas in a logic do not have an "identity" (e.g., if two modules provide the same property then they are indistinguishable) has to be taken into account when encoding well-formedness constraints related to name clashes or double definitions.

Finally, the design of the information hiding discipline (the $\hat{\mathcal{F}}$ set) influences what kinds of well-formedness constraints can be retroactively imposed on a module, since retroactive constraints (that are not anticipated by importing the module containing the checks) can only operate on the interface of the module. We have seen that this can be a limitation in the context of pluggable type systems.

6.3.3 Alignment with the Object Language

As discussed in Sec. 3.5, it is the responsibility of the programmer to make sure that the constraints modeled into the system module are sufficient to make sure that no well-formedness errors arise after code generation. If the well-formedness constraints of an object language were available as a machine-readable specification (rather than being burried in a compiler or an informal language specification), then it might be possible to design a frontend for the language the specification is written in, such that the checks agree by construction with the specification.

6.3.4 No Silver Bullet

The fact that program generators can seemingly be checked so easily once and for all may seem to be a bit suspicious. However, logic solvers are no silver bullet to the problem, and the fact that program generators can be checked is not due to some black magic but due to a careful declaration of the requirements on the input of the program generators.

To illustrate this point, consider a method generator, which takes a method body B (or other statement) and produces a method that is equivalent to

```
void execNtimes(int n) {
  for (int i=0; i<n; i++) {B}
}
```

How could this generator be proved safe, once and for all? B might have open variables that must be available in the context. B may declare itself a variable i or n which clash with the loop variable or the method argument, respectively. B may itself not even be a well-typed statement.

The answer to this question is that the REQUIRES interface of this code generator must be derived from the definition of the typing rule of the `for` statement. For better readability, we will not use Prolog syntax in the following, but standard type system notation. The (simplified) typing rule for `for` loops will typically be something like

$$\frac{\Gamma \vdash e : \texttt{int} \quad \Gamma \cup \{i : \texttt{int}\} \vdash c : \texttt{bool} \quad \Gamma \cup \{i : \texttt{int}\} \vdash b : \texttt{void}}{\Gamma \vdash \texttt{for (int } i = e; c; i\texttt{++) } \{b\} : \texttt{void}}$$

and in $\mathcal{F}_{P+}$ this typing rule would be encoded as a universally quantified implication. Obviously, in the specific example above, the body has to fulfill the constraint $C = \{i : \texttt{int}, n : \texttt{int}\} \vdash B : \texttt{void}$, hence the program generator will only be accepted by the system if C is demanded as a constraint on the input of the program generator.

Hence, in general, to prove program generators correct, one has to look at the proof tree that the solver will attempt to generate, check what the solver will try to prove about the input parameters, and add these properties to the REQUIRES interface of the program generator.

7. Related Work

Our work is related to three different areas of work: program generation, module systems and separate compilation, and logic programming.

7.1 Program Generation

There is a wide spectrum of program generation and metaprogramming approaches, such as macros, C++ templates, meta-object protocols, open compilers, or logic metaprogramming, the latter being the starting point of this work. Typically, these approaches are not modular, and checking only takes place on generated programs. Logic metaprogramming was investigated in detail in the PhD theses of [14] and [39]. Despite its generality and expressiveness, there have not been many follow-up works, although more recently several forms of logic metaprogramming have become popular again as program query languages [18, 23, 34].

There are various approaches to make certain forms of static metaprogramming safe [17, 19, 25–27]. CJ [26] offers statically safe conditional declarations. CTR [19] as well as safegen, MJ and MorphJ [25, 27, 28] offer forms of statically safe iteration over, say, the methods of a class, which can then be used to generate various forms of wrappers. These works are tailored towards a specific object language and a specific form of metaprogramming. Hence our work is not a "competitor"; rather, our formal model provides a common foundation for these works in that it characterizes the relation between modular checking and expressiveness of the metalanguage. We believe that each of these languages corresponds to a particular program representation, system module, and logic in our model. Our logic $\mathcal{F}_{P+}$ is more expressive than any of these approaches, albeit at the price of an undecidable solver.

Staging [38], such as in MetaML or MetaOCaml, can also be seen as a form of statically safe program generation, but staging does not increase the expressiveness of the language; it is mainly a technique for program specialization.

7.2 Module Systems and Separate Compilation

There is a large body of literature on module systems in both functional and object-oriented programming, but few of these works deal with the problem of incorporating metaprogramming into the module system.

The work of Cardelli [8] was the first to formally investigate the issue of separate checking, and his formal model was an important inspiration for this work. In fact, this whole work started with the idea of what happens if one revisits Cardelli's definitions and generalizes name-based imports/-exports to arbitrary formulas in a logic. Sec. 6.1.1 makes this connection explicit.

Ancona et al. [1, 2] discuss the problems of modular checking, separate compilation and linking in the context of the Java programming language. In order to cope with inter-module dependencies, they introduce a special kind of bytecode, called *polymorphic bytecode*, which can be used to infer constraints on possible linking contexts. This can be seen as an interface which consists of formulas described in a non-trivial logic. Their approach is tailored to separate checking of Java and hence complementary to our work. However, the idea to infer the requirements from the code rather than writing them down explicitly would also be interesting in our setting.

Many module or type systems allow some form of interface parameterization, such as generics in Java, SML functors, or typed λ-calculi where types can be parameterized by types (higher-order types) or values (dependent types). In such languages, the "matching" process between required and provided interfaces also involves some form of inference or, more generally, computation. However, these forms of parameterization do typically not involve computations over reifications of programs, which is the main source of the expressive power of metaprogramming.

The idea to express module consistency and compatibility by means of logic validity and consistency can also be found in other works on interface languages such as [10, 13, 20], but none of these works deal with the problem of modular metaprogramming.

7.3 Logic and Logic Programming

There is a large amount of work on module systems for logic programming, such as the module system for Prolog from [7], but all these approaches use predicate *names* and *properties* (such as arity, argument types, modes) as the basic building blocks of interface specifications, but not the logic itself as in our approach. Although module systems have been proposed for metaprogramming in logic [24, 29], these approaches are either based on names of language symbols (or declarations) and relations are usually expressed as parametrization of declarations over names or signatures, or use features that modularize the definition of predicates, but not of arbitrary logic formulas.

In the context of logic meta programming, De Volder et al. [16] propose a generic component model which is syntactically similar to our modules in that a component consists of a REQUIRES and PROVIDES interface together with an implementation, which is a list of clauses. In contrast to our framework, De Volder et al.'s component model lacks support for static checking of well-formedness and instead focuses on composition. Therefore, the approach can not be compared to the framework developed in this paper.

As described earlier, our logic $\mathcal{F}_{P+}$ is a straightforward extension of (pure) Prolog with universally quantified goals and implications as goals - techniques well-known from λProlog [33]. Our treatment of inconsistency via a designated $\perp$ constant is also standard [31].

In our future work, we would like to explore the use of other logics, such as full λProlog, as MDL in more detail. For this work, our main goal in the design of $\mathcal{F}_{P+}$ was to keep it as simple as possible, such that the description of this specific logic does not distract from our general module framework, which we consider the main contribution of this work.

Institutions [22] are similar to our axiomatization of module description logics in that both are abstractions over a range of concrete logics. However, the results in the literature on institutions are mainly about the problem of composing different logics, whereas our work is about the composition of modules that are formulated in the same logic.

8. Conclusions and Future Work

We have shown that it is possible to tame the tiger: Expressive static metaprogramming via LMP can be reconciled with separate checking. We have presented a formal framework which identifies very general conditions under which separate checking is sound, and have described the implementation of a powerful metaprogramming system which has been designed according to the formal framework.

Our future work will concentrate on four areas: First, we want to explore the trade-off between openness and separate checking in more detail. Second, we want to model existing safe metaprogramming systems such as those described in the previous section in our work. We believe that the design of an MDL and program representation for these approaches will give new insights into their expressiveness and makes it easier to compare safe metaprogramming approaches. Third, we want to investigate a larger space of possible logics and their utility for metaprogramming, such as full first or higher-order logic. Fourth, we want to study the relation between the expressiveness of type systems viewed as a logic (Curry-Howard isomorphism) and MDLs.

References

[1] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *Proceedings of the 32th Symposium on Principles of Programming Languages (POPL '05)*, New York, NY, USA, 2005. ACM Press.

[2] D. Ancona, G. Lagorio, and E. Zucca. Flexible type-safe linking of components for Java-like languages. In *Proceedings of the 7th Joint Modular Languages Conference (JMLC '06)*, volume 4228 of *Lecture Notes in Computer Science*, pages 136–154, Berlin, Heidelberg, 2006. Springer Verlag.

[3] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Proceedings of the 21st conference on Object-*

Oriented Programing, Systems, Languages, and Applications (OOPSLA '06), pages 57–74, New York, NY, USA, 2006. ACM Press.

[4] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Software Product Line Conference (SPLC '05)*, volume 3714 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2005. Springer Verlag.

[5] David Benavides, Pablo Trinidad, and Antonio Ruiz-cortés. Automated reasoning on feature models. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE '05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503, Berlin, Heidelberg, 2005. Springer Verlag.

[6] Gilad Bracha. Pluggable type systems, 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[7] Daniel Cabeza and Manuel Hermenegildo. A new module system for Prolog. In *Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Computer Science*, pages 131–148, Berlin, Heidelberg, 2000. Springer Verlag.

[8] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '97)*, pages 266–277, New York, 1997. ACM Press.

[9] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 01(1):146–166, 1989.

[10] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, Marcin Jurdzinski, and Freddy Y. C. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, pages 428–441, Berlin, Heidelberg, 2002. Springer Verlag.

[11] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the 10th conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA '95)*, pages 285–299, New York, 1995. ACM Press.

[12] Shigeru Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP '00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336, Berlin, Heidelberg, 2000. Springer Verlag.

[13] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-9)*, pages 109–120, New York, USA, 2001. ACM Press.

[14] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[15] Kris De Volder. Implementing design patterns as declarative code generators. `http://www.cs.ubc.ca/~kdvolder/publications/design_patterns-abstract.htm`, 2001.

[16] Kris De Volder, Johan Fabry, and Roel Wuyts. Logic meta components as a generic component model. In *Fifth International Workshop on Component-Oriented Programming, Workshop reader of ECOOP '00*, 2000.

[17] Dirk Draheim, Christof Lutteroth, and Gerald Weber. A type system for reflective program generators. In *Proceedings of the 4th international conference on Generative programming and component engineering (GPCE '05)*, Lecture Notes in Computer Science, pages 327–341, Berlin, Heidelberg, 2005. Springer Verlag.

[18] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering (ICSE '08)*, pages 391–400, New York, NY, USA, 2008. ACM Press.

[19] Manuel Fähndrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In *Proceedings of the 5th international conference on Generative programming and component engineering (GPCE '06)*, pages 275–284, New York, NY, USA, 2006. ACM Press.

[20] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-9)*, pages 229–236, New York, NY, USA, 2001. ACM Press.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, Indianapolis, USA, January 1995.

[22] J.A. Goguen and R.M. Burstall. Introduction to institutions. In *Logic of Programs, Workshop, Carnegie Mellon University*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer Verlag, 1983.

[23] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In *Proceedings of the 20th European conference on Object oriented programming (ECOOP '06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Heidelberg, 2006. Springer Verlag.

[24] P. Hill. A module system for meta-programming. *Logic Program Synthesis and Transformation – Meta-Programming in Logic*, 883:395–409, 1994.

[25] Shan S. Huang and Yannis Smaragdakis. Expressive and safe static reflection with MorphJ. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI '08)*, pages 79–89, New York, NY, USA, 2008. ACM Press.

[26] Shan S. Huang, David Zook, and Yannis Smaragdakis. cJ: enhancing Java with safe type conditions. In *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD '07)*, pages 185–198, New York, NY, USA, 2007. ACM Press.

[27] Shan S. Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely Shaping a Class in the Image of Others. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP '07)*, Berlin, Heidelberg, 2007. Springer Verlag.

[28] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with safegen. In *Proceedings of the 4th international conference on Generative programming and component engineering (GPCE '05)*, Lecture Notes in Computer Science, pages 309–326. Springer Verlag, 2005.

[29] Evelina Lamma and Paola Mello. Modularity in logic programming. In *Proceedings of the eleventh international conference on Logic programming*, pages 15–17, Cambridge, MA, USA, 1994. MIT Press.

[30] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. In *Journal on Expert Systems with Applications*, pages 236–243, New York, NY, USA, 2001. Elsevier Science Inc.

[31] Dale Miller. A logical analysis of modules in logic programming. *J. Log. Program.*, 6(1-2):79–108, 1989.

[32] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solutions. *J. Log. Program.*, 25(2):119–161, 1995.

[33] Gopalan Nadathur and Dale Miller. An overview of λProlog. In *Proceedings of the Fifth International Logic Programming Conference*, pages 810–827, Cambridge, MA, USA, 1988. MIT Press.

[34] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP '05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240, Berlin, Heidelberg, 2005. Springer Verlag.

[35] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[36] Raymond Reiter. On closed world databases. In *Logic and Databases*, pages 55–76, New York, NY, USA, 1978. Plenum Press.

[37] Ehud Shapiro and Leon Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, Cambridge, MA, USA, April 1994.

[38] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 symposium on Partial Evaluation and semantics-based Program Manipulation (PEPM '97)*, pages 203–217, New York, USA, 1997. ACM Press.

[39] Roel Wuyts. *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

APPENDIX

A. Proofs

We will need the following lemmas for the proofs:

LEMMA 22 (Monotonicity of ifc). *If ifc(F) and $F' \subseteq F$, then ifc(F').*

Proof. Assume that ifc(F) and $\neg$ifc(F'), i.e., $(\exists I \subseteq \hat{\mathcal{F}})\ F' \cup I \vdash \bot$. By monotonicity of $\vdash$ it follows that $F \cup I \vdash \bot$ and therefore $\neg$ifc(F), which is a contradiction. ■

LEMMA 23. $\approx_\vdash$ *is an equivalence relation.*

Proof. We have to show:

Reflexivity Since $R_1 = R_2$, we have to show that $P \cup R \vdash R$, which follows from the monotonicity of $\vdash$.

Symmetry The definition of $\approx_\vdash$ is symmetric.

Transitivity Both $=$ and $\vdash$ are transitive.

■

LEMMA 24. $\forall k \in \mathbb{N}, M, M'\ (M \rightsquigarrow^k M' \Rightarrow M \approx_\vdash M')$

Proof. The case $k = 0$ is trivial. To prove $k = 1$ we assume that $(R, I, P) \rightsquigarrow (R \setminus \{f\}, I, P)$.
Now we have to prove that $R \cup P \vdash R \setminus \{f\}$, which follows from 1.4 and 1.3, and $R \setminus \{f\} \cup P \vdash R$, which is true, because $R \setminus \{f\} \vdash R \setminus \{f\}$ and $P \vdash \{f\}$.
The case $k > 1$ follows from Lemma 23 (transitivity of $\approx_\vdash$). ■

A.1 Theorem 7

Proof. Let $I = \cup_{i \in \{1,\ldots,n\}} I_i$ be the implementation and $P = \cup_{i \in \{1,\ldots,n\}} P_i$ the PROVIDES part of the composed module. By 5.10, we can split I into two disjunct sets $I = I' \cup P'$, where $I' \subseteq \hat{\mathcal{F}}$ and $P' \subseteq P$. Because all modules are compatible, we have ifc(P) and by Lem. 22 we get ifc(P'), which means that P' is consistent together with arbitrary implementation formulas, and in particular it is consistent with I'. From this follows that $I \nvdash \bot$.

■

A.2 Lemma 9

Proof. By definition of $\rightsquigarrow$, M' is valid. If M' were inconsistent, then M would have been inconsistent, too, by Lemma 22. But M is consistent by assumption, hence M' must be consistent. ■

A.3 Theorem 12

Proof. Let $M = (R_1 \cup R_2, I_1 \cup I_2, P_1 \cup P_2)$. Since M' and M'' are minimal reductions of M, we have $M \rightsquigarrow^k M'$ and $M \rightsquigarrow^l M''$. By Lemma 24 it follows that $M \approx_\vdash M'$ and $M \approx_\vdash M''$ and by transitivity and symmetry of $\approx_\vdash$ we have $M' \approx_\vdash M''$. ■

A.4 Theorem 14

Proof. Let $M_i = (R_i, I_i, P_i)$ for $i = 1, 2$. Let $M = (R_1 \cup R_2, I_1 \cup I_2, P_1 \cup P_2)$ as in the first step in Def. 10. Then M is valid because for each $f \in P_1 \cup P_2$ we have $R_1 \cup R_2 \cup I_1 \cup I_2 \vdash f$ by the validity of M_1 and M_2 and the monotonicity of $\vdash$ (Def.1.3). $(R', I', P') = (R', I_1 \cup I_2, P_1 \cup P_2) = M_1 \oplus_\sigma M_2$. M is also consistent because ifc$(R_1 \cup R_2 \cup P_1 \cup P_2)$ by $M_1 \div M_2$. By Lemma 9 consistency is preserved in every reduction step, hence $M_1 \oplus_\sigma M_2$ must also be consistent. ■

A.5 Theorem 15

Proof. Let $M_i = (R_i, I_i, P_i)$ for $i \in \{1, 2, 3\}$ and $M_1 \oplus_\sigma M_2 = (R, I, P)$. By Def. 8 and 10, $P = P_1 \cup P_2$ and $R \subseteq R_1 \cup R_2$. The theorem now follows by Def. 6 and Lemma 22. ■

A.6 Theorem 16

Proof. Let $(M_i)_{1\le i\le 3} = (R_i, I_i, P_i)$, then

$$\begin{aligned}
&(M_1 \oplus_\sigma M_2) \oplus_\sigma M_3\\
&\qquad \approx_\vdash (I_1 \cup I_2, R_1 \cup R_2, P_1 \cup P_2) \oplus_\sigma M_3\\
&\qquad \approx_\vdash (I_1 \cup I_2 \cup I_3, R_1 \cup R_2 \cup R_3, P_1 \cup P_2 \cup P_3)\\
&\qquad \approx_\vdash M_1 \oplus (I_2 \cup I_3, R_2 \cup R_3, P_2 \cup P_3)\\
&\qquad \approx_\vdash M_1 \oplus_\sigma (M_2 \oplus_\sigma M_3)
\end{aligned}$$

■

A.7 Theorem 17

Proof. By induction on n. The base case $n = 1$ is trivial, the case $n = 2$ is covered by Thm. 14. For the inductive case $n \ge 3$, let $M = \bigoplus^\sigma_{1\le i\le n-1} M_i$. By induction hypothesis we know module-ok(M). By applying Thm. 15 $n-2$ times, we get $M \div M_n$. Now the theorem follows by applying Thm. 14 to M and M_n. ■

A.8 Theorem 20

Proof. We prove the parts of the theorem in the order as stated.

- A reduction step removes exactly one formula from the REQUIRES set of the module, and the REQUIRES set is finite.
- Follows by Thm. 17.
- From Thm. 12 we know that $M' = \bigoplus^\sigma_{1\le i\le n} M_i \approx_\vdash \bigoplus^{\sigma'}_{1\le i\le n} M_i = M''$. By transitivity of $\approx_\vdash$ we know that $M' \approx_\vdash M''$ and by the definition of $\rightsquigarrow$, their implementation and PROVIDES part are equal. Thus, the only way it could happen that Final$(M') \neq$ Final(M'') is when one finalization fails while the other succeeds. But because $M' \approx_\vdash M''$, there is no requirement formula, that can be proved for one module but not for the other.
- The implementation part of M is the union of the implementation parts of the modules $M_1, \dots, M_n$ and thus consistent by Thm. 7.
- If M is valid, and $M \rightsquigarrow^{\text{CW}}_{\text{min}} M'$ then M' is valid by definition of $\rightsquigarrow_{\text{min}}$, that is, $R \cup I \vdash P$. By the definition of Final, M' has the form $(\emptyset, I, P)$, thus $I \vdash P$.

■

A.9 Theorem 21

Before proving the theorem, we have to define T.

DEFINITION 25.

$$T(l) = \begin{cases} l & l \in L \\ \textit{true} & l \in \hat{L} \end{cases}$$

$$T((\forall v)\ f) = (\forall v)T(f)$$

$$T((\exists v)\ f) = (\exists v)T(f)$$

$$T(l \leftarrow l_1, \dots, l_n) = T(l) \leftarrow T(l_1), \dots, T(l_n)$$

$$T(\{l_1, \dots, l_n\}) = \{T(l_1), \dots, T(l_n)\}$$

Proof. Let Γ be arbitrary but fixed.
$\Rightarrow$ Assume that $\exists F \subseteq \hat{\mathcal{F}}_{P+}$ such that $\Gamma \cup F \vdash \bot$. We observe that a proof of $\bot$ does not include proofs of implications. Thus, the proof tree contains only substitutions of literals by bodies of matching clauses, eliminations of existential quantification and substitution of constants for universally quantified variables. By replacing every node of the tree that contains implementation literal with *true* and removing their subtrees, we end up with another valid proof tree. All the substitutions of clause bodies that are in the tree can be done using formulas from $T(\Gamma)$, which means that the modified tree is a proof tree for $\bot$ in $T(\Gamma)$, and thus $T(\Gamma) \vdash \bot$.
$\Leftarrow$ Assume that $T(\Gamma) \vdash \bot$. We can choose $F = \hat{L}$, for which $\Gamma \cup F \vdash \bot$, because all implementation literals that can be used in the proof are true. ■