Towards Modular Extensions for a Modular Language

facultad de informática
universidad politécnica de madrid

J. F. Morales
M. Hermenegildo
R. Haemmerlé

January, 2011

Authors

Jose F. Morales[1], Manuel V. Hermenegildo[1,2], and Rémy Haemmerlé[2]
[1] IMDEA Software Research Institute, Madrid (Spain)
[2] School of Computer Science, T. U. Madrid (UPM), (Spain)
{josef.morales,manuel.hermenegildo}@imdea.org
remy@clip.dia.fi.upm.es, herme@fi.upm.es

Keywords

Compilation; Modules; Modular Program Processing; Separate Compilation; Prolog; Ciao.

Abstract

Modularity allows the construction of complex designs from simpler, independent units that most of the time can be developed separately. In this paper we are concerned with developing mechanisms for easily implementing modular extensions to modular (logic) languages. By (language) extensions we refer to different groups of syntactic definitions and translation rules that extend a language. Our application of the concept of modularity in this context is twofold. We would like these extensions to be modular, in the above sense, i.e., we should be able to develop different extensions mostly separately. At the same time, the sources and targets for the extensions are modular languages, i.e., such extensions may take as input separate pieces of code and also produce separate pieces of code. Dealing with this double requirement involves interesting challenges to ensure that modularity is not broken: first, combinations of extensions (as if they were a single extension) must be given a precise meaning. Also, the separate translation of multiple sources (as if they were a single source) must be feasible. We present a detailed description of a code expansion-based framework that proposes novel solutions for these problems. We argue that the approach, while implemented for Ciao, can be adapted for other languages and Prolog-based systems.

Contents

1 Introduction

The choice of a good notation and adequate semantics when encoding a particular problem can dramatically affect the final outcome. An extreme example are programming pearls whose beauty is often completely lost when translated to a distant language. In practice, large projects are bigger than pearls and often no single language fulfills all expectations (which can include many aspects, such as development time or execution performance). The programmer is forced to make a commitment to one language —and accept sub-optimal encoding— or more than one language —at the expense of interoperability costs.

An alternative is to provide new features and idioms as syntactic and semantic extensions of a language, thus providing the notational convenience while avoiding inter-language communication costs. Macros have been used traditionally in many languages (such as, e.g., C, C++, or Lisp) as a way to extend the syntax and optimize the code. Despite being a powerful language extension mechanism, their pure syntactic nature and poor semantic integration with the underlying language have made them a generally despised concept that is as a result being avoided in many newly designed languages. Some of their uses are being replaced by tamed features such as, e.g., for C++, function inlining or code templates. In the case of Prolog, expansion systems have traditionally offered a quick way to develop variants of logic languages and semantics, and to enhance program representation. These facilities have been used to develop many language extensions such as experimental domain specific languages, constraint systems, optimizations, debugging tools, etc.

A more elaborate extension system than that offered by traditional Prolog systems is implemented in Ciao [CH00, HBC+10]. Ciao allows defining extension units called *packages*, which are custom source files containing syntactic definitions (e.g., new operators), the declaration of language extension hooks, and the dependencies to *compilation modules* (which define the actual extension code for each hook). The fundamental novelty of the Ciao approach w.r.t. traditional Prolog systems is the separation between translation code (executed at compile-time) and source code (executed at run-time), which eliminates the uncertainty with regards to when expansions are applied, as well as providing a richer set of hooks. The code in packages is distributed among several files. The main file defines the syntax of the language, e.g., new operators and new declarations. It also includes declarations that point to the expansion module, containing compile-time code which defines the *hooks* called by the compiler (optionally expansions can also be defined to be applied to read operations at run time). This system has been used to implement new dialectic extensions such as DCGs, languages (e.g., CHR), code instrumentation (e.g., profiling, debugging), optimizations (e.g., unfolding), etc.

The Ciao approach has been successful at adapting the Prolog notion of expansions to a system where full separate compilation is available. However, it also has some shortcomings that have become apparent over time and which we discuss in the following.

Combination of Extensions. In practice it is often the case that more than one extension is desired for the same module (for example functional notation, DGCs, and profiling). Determining if the application of such a set of extensions on the same source will have the expected results is complicated, since, among other issues, the application order is often relevant. The user has to control the order in which *packages* are applied by enumerating them in the correct sequence in order to obtain the desired effect. In other Prolog systems [Wie10] where language extensions are enabled by modules (using the `use_module` directive), the problem is aggravated, since then the order in which modules are imported becomes relevant. Thus, a solution for the problem of

combining extensions that does not involve explicit ordering by the user is very desirable and would benefit systems taking both of the approaches mentioned above.

In some cases, complex extensions are designed to work on monolithic code units (e.g., CHR to Prolog) and this often means that many combinations, although conceptually feasible, are not allowed in practice. For example, suppose that we would like to combine *functional notation* [CCH06] and CHR [Fw09] to use functional syntax in the rule constraints. Unfortunatelly, code like:

```
Y = ~neg(X) \ Z = ~and(X,Y) <=> Z=0.
```

cannot be translated to

```
neg(X,Y) \ and(X,Y,Z) <=> Z=0.
```

since we would like to normalize functional syntax before the CHR translation, but there is no way to indicate to the functional expansion that the constraints have to be treated syntactically as goals (that should be defined by CHR).[1]

Module-aware Extensions The integration with the underlying module system is weak: e.g., it is not straightforward to determine during expansion to what module a goal being expanded belongs, or to export new declarations. For example, any the `:- argnames` declarations (from the *named records* package) are only visible locally. It is thus necessary to make the extensions module aware, while at the same time constraining them to respect the module system. It is well known that modularity, if not designed carefuly, can make static analysis impossible [Car97]. A flexible extension system that however allows breaking modularity renders any efforts towards static transformations useless.

This paper presents a number of novel contributions aimed at addressing these problems. The context of our work is that of dynamic languages, where code can be loaded at any time during execution, and thus there is no explicit distinction between the compile- and run-time phases. We adopt the Ciao [CH00] approach to conciliating dynamic features with static techniques (separate compilation, analyisis, etc.), where code is separated in modules, compilation is *hidden* from the user, but there exist *module invariants* that constrain the dynamic features when required. We contribute a formal description of the compilation model, that includes the loading of compilation modules and its use during compilation of other modules. We discuss based on this model the different phases and the information that is treated and available at each phase. These considerations will determine the expressiveness required from the extension rules. Based on this, we propose a set of rule-based extension mechanisms that we argue generalizes previous approaches and allows us to provide better solutions for a good number of the problems mentioned earlier.

The paper is structured as follows. Section 2 provides a formal specification of the compilation model. Section 3 gives a detailed description of the translation process for extensions. We consider this an appropiate abstraction for the problem. However, the final implementation language does not necessarily need to follow this style. Section 4 illustrates the rules defined in the previous section by defining several language features and extensions. We close with a discussion of related and future work in Section 5, and conclusions in Section 6.

[1]The CHR translation could invoke the functional syntax translation directly. However that breaks the mechanisms for language extension. This necessity motivates our work.

2 A Modular Compilation Model

2.1 Separate Compilation in a Dynamic Language

Program execution can be specified and performed directly on the source code, but in practice most languages require some preprocessing (at least parsing). In a static and monolithic setting, the simplest trace from source to execution can be formulated as follows:

$$\langle s_0 \rangle \ \mathsf{comp}(p, m) \fatsemi \mathsf{load}(m) \fatsemi e \ \langle s \rangle$$

where: $\fatsemi$ is relation composition ($s(R \fatsemi S)t \equiv sRs' \wedge s'St$), $\mathsf{comp}(p, m)$ relates the input and output states during compilation of program p, storing the results in the state and identifying them by m, load loads m in the state, and e is some arbitrary code invoking code from m.

Although this scheme has advantages (like easily supporting whole-program optimizations), it is also often costly and impractical. Even when the program is designed as distinct parts that are updated separately, compilation and loading require processing the whole source. In the case of *separate compilation*, advantage can be taken of the logical decomposition of programs into smaller units (modules) to perform compilation and load individually. Let m_i be each of the unique identifiers for modules (the *module name*) and p_{m_i} the source code for each of them. The modules are compiled and then loaded separately (or linked and loaded) before the program execution takes place:

$$\langle s_0 \rangle \ \mathsf{comp}(p_{m_1}, m_1) \fatsemi \ldots \mathsf{comp}(p_{m_n}, m_n) \fatsemi \mathsf{load}(m_1) \fatsemi \ldots \mathsf{load}(m_n) \fatsemi e \ \langle s \rangle$$

This allows reduced compilation, analysis, and preprocessing time, and the distribution of precompiled libraries. In the case of a *dynamic* language, module compilation and loading can happen in the middle of the execution. The execution trace, that can be extended indefinitely, will have the form:

$$\langle s_0 \rangle \ \ \ldots e_{i-1} \fatsemi \mathsf{comp}(p_{m_i}, m_i) \fatsemi \mathsf{load}(m_i) \fatsemi e_i \ldots \ \langle s \rangle$$

Since the value of the module source p_{m_i} may be dynamic too (i.e., dependent on the execution state or environment), in principle the only way to ensure in all cases that we are loading the latest source is by preceding each load by a compilation. A way to overcome this limitation is to keep in the state the compilation results[2] and only recompute them when necessary. In order to define the conditions for *re*compilation and what information can be saved between compilations, we define a set of *compilation equations* in Section 2.2.

Supporting Dynamic Semantics

We start from the observation that in dynamic languages, not only arbitrary programs can be loaded at runtime, but also their contents may change during execution. Thus, we need to separate the invariant part from the dynamic one:

Definition 2.1 (Module Invariant) The *module invariant* of module i is the set of definitions (e.g., predicate code and properties, imported modules) that will not change during the execution, once it has been loaded. Relaxed invariants allow more *dynamic* programs, tigher invariants allow more aggressive compilation techniques. □

[2] In a realistic scenario, persistent storage such as disk would be included as part of the state.

In order to support fully dynamic semantics, each module has an associated state. E.g., for a *dynamic predicate*, the *invariant* states the initial state for the predicate, and thus gives the initial values for the dynamic interface, while asserting a clause modifies the module state.

2.2 Compilation Equations

We will assume for programmer convenience that the use of properties bound to an identifier is allowed to precede the definition of the identifier.[3] This implies that the compilation of a single module needs at least two global phases. In the first phase, the source code (for brevity denoted as p_i) is read, parsed, and normalized. The information about local definitions (e.g., defined predicates) and the module interface (e.g., exported predicates) is available only at the end of this phase. We call this the *syntactic* phase, since we are not yet able to fully provide a semantics for the program, which may depend on external definitions from imported modules.[4] In the second phase, the interface of the imported modules is collected and processed alongside with the normalized code. The output, which we say is in the *kernel* language, can be passed to the input of the code generation phase to generate executable object code (e.g., bytecode) or treated by program analysis tools. We name this the *semantic* phase. Additionally, the use of *compilation modules* in each module can be declared to define custom transformations in both the first and second phase.

Definition 2.2 (Parsed Module) Reading a source p_i yields a *parsed module* ϖ_i, a set of propositions including the contents of the module. It will contain at least the dependencies and code, where judgement $\varphi_i \vdash \mathsf{use\text{-}mod}(j)$ states that module j is used by module i, $\varphi_i \vdash \mathsf{use\text{-}cmod}(j)$ that j is a compilation module required for the compilation of i, and $\varphi_i \vdash \mathsf{sents}(s_1 \ldots s_n)$ specifies the sequence of module sentences (clauses and declarations). We will identify the parsed module just after read as ϖ_i^0, while ϖ_i will be the final normalized value. □

Definition 2.3 (Interface Projection) Given a parsed module ϖ_i its *interface projection* φ_i (or simply *interface*) denotes the subset of exported properties (e.g., exported predicates), that are visible from importing modules. □

Definition 2.4 (Dependency Relations) We define the following shorthands for *dependency relations* between modules i and j:

$$i \xrightarrow{d} j \equiv (\varphi_i \vdash \mathsf{use\text{-}mod}(j)) \tag{1}$$

$$i \xrightarrow{c^0} j \equiv (\varpi_i^0 \vdash \mathsf{use\text{-}cmod}(j)) \tag{2}$$

$$i \xrightarrow{c} j \equiv (\varphi_i \vdash \mathsf{use\text{-}cmod}(j)) \tag{3}$$

where Eq. (1) is the dependency due to module i using module j, and Eq. (3) the dependency due to module i requiring compilation module j. Eq. (2) is equivalent to Eq. (1), except that the former is known at the moment when the module is read (required in order to know the compilation modules that alter the normalization process), while the latter is in the module interface. □

[3] Unlike in languages like C, where functions must be defined before its use.

[4] It is important not to confuse *importing* a module with *including* a file. The latter is purely syntactic and can be performed during program reading. For the sake of clarity, we omit dependencies to included files in further sections.

Definition 2.5 (Link Set) The *link set* of a module is defined as the reflexive transitive closure of the relation $\xrightarrow{d}$, that we will denote as $(\xrightarrow{d\star}) \triangleq (\xrightarrow{d})^\star$. The link set of a module i is the minimum set of additional modules that are required during execution.

Definition 2.6 (Compilation Equations) The equations that describe all the compilaton phases for each module i, explained below, are the following:

$$\varpi_i^0 = \mathsf{parse}(p_i) \wedge \varpi_i = \mathsf{comp\text{-}syn}(\varpi_i^0) \wedge \varphi_i = \pi_{\mathsf{itf}}(\varpi_i) \tag{4}$$

$$\omega_i = \mathsf{comp\text{-}sem}(\varpi_i \cup \{\mathsf{itf}(j, \varphi_j) \mid i \xrightarrow{d} j\}) \tag{5}$$

$$\omega_i^\star = \{\omega_j \mid i \xrightarrow{d\star} j\} \tag{6}$$

where:

$$\mathsf{comp\text{-}syn} \triangleq \mathsf{link}[\![\mathsf{syn\text{-}tr}, \bigcup_{i \xrightarrow{c^0} j} \omega_j^\star]\!] \tag{7}$$

$$\mathsf{comp\text{-}sem} \triangleq \mathsf{link}[\![\mathsf{sem\text{-}tr}, \bigcup_{i \xrightarrow{c} j} \omega_j^\star]\!] \tag{8}$$

Eq. (4) defines the syntactic phase, which reads p_i as ϖ_i^0, transforms it to ϖ_i, and projects φ_i as the interface. The semantic phase is defined in Eq. (5), which takes as input the parsed module ϖ_i enlarged with the interface information of imported modules. We will abbreviate judgements on imported interfaces as $(\varpi_i \vdash_j P) \equiv (\varpi_i \vdash \mathsf{itf}(j, \varphi_j)) \wedge (\varphi_j \vdash P)$. Finally, Eq. (6) describes not a real phase, but a fake variable that denotes the link set. This set is the input for module linking (i.e., runtime loading or generation of executables), which happens in Eq. (7) and Eq. (8). These last two definitions describe, respectively, the result of dynamically linking (denoted as $\mathsf{link}[\![f, \cdot]\!]$) the syntactic and semantic translation code (part of the compiler) with the required compilation modules (a process that happens at compilation time). Figure 1 shows the processes required for the incremental compilation of a module i depending on module j and compilation module module k.

Compilation Module Loops and Bootstrapping In general, we cannot provide a solution if $i\ (\xrightarrow{c}\xrightarrow{d\star})^\star\ i$, i.e., if a module i uses a compilation module that depends on i, or another module which depends on a compilation module that depends on i, etc. These loops cause a *deadlock* situation, where the compilation module cannot be compiled because it requires its compiled code beforehand. However, it is common to have languages that must compile themselves. We solve the issue by distinguishing between normal and static modules. Static modules have been compiled previously and their ω_i and φ_i are kept for following compilations (say ω_i^S and φ_i^S respectively). In that case, $(\varphi_i = \varphi_i^S \wedge \omega_i = \omega_i^S)$. The set of all static modules for the compiler constitutes the *bootstrap* system. Note that *self-compiling* modules require caution, since accidentally losing the bootstrap will make the source code useless.

Proposition 2.7 (Compilation Dependencies) The previous equations showed how each non-source variable is computed. Every time some input is modified, the changes need to be propagated through all the variables until all of them have been updated. The graph of dependencies between the variables in the equations in Definition 2.6 has the following edges

(ϖ_i^0 does not need to be kept; we can collapse edges by removing the node):

$$\varpi_i \dashleftarrow p_i \qquad\qquad \varpi_i \dashleftarrow \omega_{k^0}^\star \text{ (for each } i \xrightarrow{c^0} k^0)$$
$$\varphi_i \dashleftarrow \varpi_i$$
$$\omega_i \dashleftarrow \varpi_i \qquad \omega_i \dashleftarrow \varphi_j \text{ (for each } i \xrightarrow{d} j) \qquad \omega_i \dashleftarrow \omega_k^\star \text{ (for each } i \xrightarrow{c} k)$$
$$\omega_i^\star \dashleftarrow \omega_l \text{ (for each } i \xrightarrow{d\star} l)$$

Note that the graph is not static since the dependency relations between modules ($\xrightarrow{d}$, $\xrightarrow{c}$, $\xrightarrow{c^0}$) depend on data obtained from the contents of p_i.

This problem is related to *self-adjusting* computations, as treated in [Aca09]. Nevertheless, we use an *ad-hoc* algorithm specialized for the shape of our graph. As a safe over-approximation to look for changes, we will use *timestamps* for some dependencies, so that each time a variable is updated its timestamp is incremented. If an input is *more recent* than an output, it will need recomputation. In order to avoid excessive recompilations, the algorithm treats interfaces specially and only triggers recomputations if their contents actually changed. The algorithm to update the compilation variables, $\langle F \rangle$ update(g) $\langle F' \rangle$, will take a variable name to be updated g, a mapping between variable names and values F, and return an updated mapping F' where g and all its dependencies has been updated.[5] An interesting expected property is that *updating* a sequence of modules must not be affected by order or repetition.

Module Invariants and Extensions Although the kernel language may provide low-level pathways if necessary (e.g., to implement debuggers, code inspection tools, or advanced transformation tools), it is important not to break the module *invariants*. One invariant is φ_j, which once computed cannot be changed without invalidating the compilation of any module i that imports it. For this reason, a *semantic* expansion cannot modify the module interface.

3 Language Extensions

We define the language extensions as translations that manipulate a symbolic representation of the module. For simplicity we will use *terms* representing abstract syntax trees, denoted by T as following the usual definition of *ground* terms in *first order logic*. For easier notation, we include sequences of terms (*Seq(T)*) as part of T.[6] We also assume some standard definitions and operations on terms: term-fn(x) denotes the $(name, arity)$ of the term, get-args : $T \to Seq(T)$ obtains the term arguments (i.e., the sequence of children), and set-args : $T \times Seq(T) \to T$ replaces the arguments of a term.

3.1 Translation Rules and Algorithm

We use a homogeneous term representation for the program, but terms may represent a variety of language elements. The interpretation of each term is given by its surrounding context. In order to reflect this, we label the input term during translation with a symbolic *kind* annotation. That annotation indicates how the term is going to be treated during transformation. We define

[5] Note that data can be kept in persistent disk storage or in memory.

[6] We will assume –for simplicity and contrary to common practice– that when compiling a program variables are read as special ground terms.

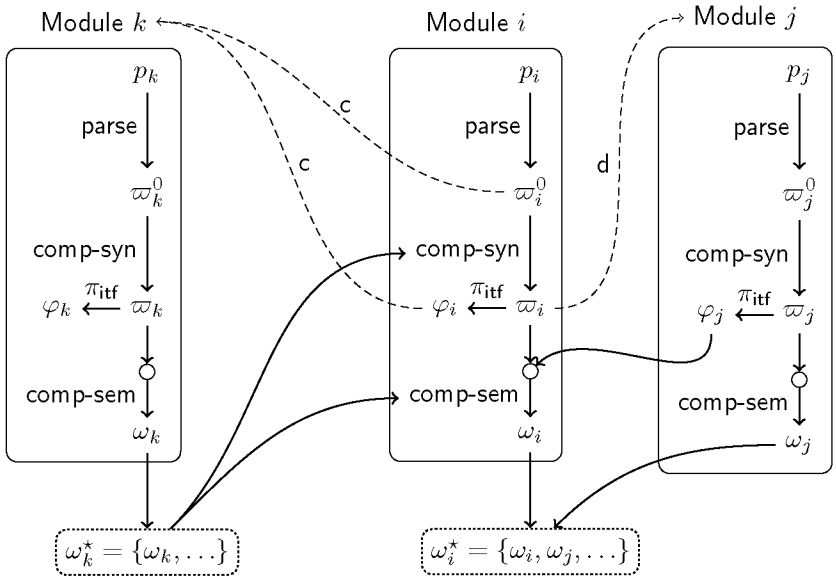

Figure 1: Compilation dependencies for module i, which imports module j, and requires compilation module k.

a main transformation algorithm $\mathbf{tr}[\![x : \kappa]\!] = x'$ (Fig. 2), which, given a term x of *kind* κ, obtains a term x' by applying the available rules. It will be called from both syn-tr and sem-tr, which appeared in Definition 2.6.

There are some *special kinds*, whose meaning is as follows: the $\mathit{try}(t, \kappa_1, \kappa_2)$ kind tries to transform the input with the relation t. If it is possible, the resulting term is transformed with kind κ_1. Otherwise, the untransformed input is retried with kind κ_2. This is useful to compose translations. The *final* kind stops translation. The $\mathit{seq}(\kappa)$ kind indicates that the input term is a sequence of elements of kind κ.[7] The remaining transformations are driven by rules. Note that the rules may contain guards in order to make them conditional on the term. Rule $\kappa \succ \kappa'$ denotes that every term of kind κ must be translated as κ'. Rule $x : \kappa \Longrightarrow x' : \kappa'$ is the same, but the term is modified. Finally, rules $\mathbf{decons}\lfloor x : \kappa_x \rfloor \rightsquigarrow \vec{a} : \vec{\kappa}$ and $\mathbf{cons}\lfloor \vec{a} : \vec{\kappa},\ x \rfloor \rightsquigarrow \kappa_x : x'$ allow the decompositon (*decomp*) of a term into smaller parts, which are translated and then put together (*comp*). We will see examples of all these rules later.

We divided expansions into fine-grained translations because we want them to be interleaved with other rules. We want to combine them. Monolithic expansions would render their combination infeasible in many cases.

Composition of Transformations Note that the transformation algorithm does not make any assumptions regarding the order in which rules are defined in the program, given that the rules define a fixed order relation between kinds. We will see in Section 4 how to give an unambiguous meaning to conflicting rules (targeting the same kind).

Example 3.1 (Combining Transformations) The example shown in the introduction about

[7] In the Prolog implementation sequences are replaced by lists.

$$
\begin{aligned}
\mathbf{tr}[\![x : \mathit{try}(t, \kappa_1, \kappa_2)]\!] &= \begin{cases} \mathbf{tr}[\![x' : \kappa_1]\!] & \text{if } t(x, x') \\ \mathbf{tr}[\![x : \kappa_2]\!] & \text{otherwise} \end{cases} \\
\mathbf{tr}[\![x_1 \ldots x_n : \mathit{seq}(\kappa)]\!] &= (\mathbf{tr}[\![x_1 : \kappa]\!] \ldots \mathbf{tr}[\![x_n : \kappa]\!]) \\
\mathbf{tr}[\![x : \mathit{final}]\!] &= x \\
\mathbf{tr}[\![x : \kappa]\!] &= \mathbf{tr}[\![x : \kappa']\!] \quad (\text{if } \kappa \succ \kappa') \\
\mathbf{tr}[\![x : \kappa]\!] &= \mathbf{tr}[\![x' : \kappa']\!] \quad (\text{if } x : \kappa \Longrightarrow x' : \kappa') \\
\mathbf{tr}[\![x : \kappa_x]\!] &= \mathbf{tr}[\![x' : \kappa'_x]\!] \quad (\text{if } \mathbf{decons}\lfloor x : \kappa_x \rfloor \rightsquigarrow \vec{a} : \vec{\kappa}) \\
&\quad \text{where} \\
&\quad a'_i = \mathbf{tr}[\![a_i : \kappa_i]\!] \quad \forall i. 1 < i < |\vec{a}| \\
&\quad \mathbf{cons}\lfloor x : \kappa_x,\ \vec{a'} \rfloor \rightsquigarrow x' : \kappa'_x
\end{aligned}
$$

Figure 2: The Transformation Algorithm

merging CHR and *functional syntax*, can be solved now with rules such as:

$$
\begin{aligned}
\mathbf{decons}\lfloor (a \setminus b \texttt{ <=> } c) : \mathit{chrclause}_1 \rfloor &\rightsquigarrow (a\ b\ c) : (\mathit{sgoal}\ \mathit{sgoal}\ \mathit{sgoal}) \\
\mathbf{cons}\lfloor \_ : \mathit{chrclause}_1,\ (a\ b\ c) \rfloor &\rightsquigarrow (a \setminus b \texttt{ <=> } c) : \mathit{chrclause}_2
\end{aligned}
$$

meaning that in the middle of the translation from the hypothetical kinds $\mathit{chrclause}_1$ and $\mathit{chrclause}_2$ we allow treatment of a kind *sgoal* (which, e.g., could be treated by the functional syntax package). □

Customized Compilation Stack Note that the same mechanism described to transform terms representing pieces of code can be used to treat full program modules. E.g., for the `argnames` example, shown in the introduction, it is possible to insert additional passes that store declarations in the module interface.

4 Examples and Applications

We will illustrate the extension framework with examples and applications. First we will show an encoding of the translation hooks available in Ciao using the new translation rules. Then we will provide examples of how some non-trivial language features can be handled.

Note on Implementation For conciseness and consistency, we continue using the formal notation in all the following sections. Writing the Prolog equivalent of both the rules and the driver algorithm presented here is straightforward. As implemented in the Ciao compiler, Prolog terms can be used to represent the abstract syntax tree. The different stages of compilation can be kept in memory as facts in the dynamic database, with extra arguments to identify the module. Since the transformation approach proposed is still an experimental feature, we have not studied performance fully, but there is no reason to expect lower performance than with the previous approach. In the Ciao implementation the older *hooks* are still being preserved for backwards compatibility.

$$
\begin{aligned}
\mathit{sents} &\succ \mathit{sents}_{\vec{\mathcal{E}_s}} \\
\mathit{sents}_{ts} &\succ \mathit{seq}(\mathit{sent}_{ts}) \\
\mathit{sent}_{[t|ts]} &\succ \mathit{try}(t, \mathit{sents}_{ts}, \mathit{sent}_{ts}) \\
\mathit{sent}_{[]} &\succ \mathit{term} \\
\\
\mathit{term} &\succ \mathit{term}_{\vec{\mathcal{E}_t}} \\
\mathit{term}_{[t|ts]} &\succ \mathit{try}(t, \mathit{term}_{ts}, \mathit{term}_{ts}) \\
\mathit{term}_{[]} &\succ \mathit{rterm} \\
\textbf{decons}\lfloor x : \mathit{rterm} \rfloor &\rightsquigarrow \mathsf{get\text{-}args}(x) : (\mathit{term} \ldots \mathit{term}) \\
\textbf{cons}\lfloor x : \mathit{rterm},\ \vec{a} \rfloor &\rightsquigarrow \mathsf{set\text{-}args}(x, \vec{a}) : \mathit{final}
\end{aligned}
$$

Figure 3: Emulating `term_trans` and `sentence_trans`

4.1 Emulating Ciao (and Prolog) Translations

Let us define a *transformation specification* as the tuple $\mathcal{E} = (\mathcal{E}_t, \mathcal{E}_s, \mathcal{E}_c, \mathcal{E}_g)$, so that $\mathcal{E}_t$, $\mathcal{E}_s$, $\mathcal{E}_c$, $\mathcal{E}_g \subseteq T \times T$, and they are partial functional relations. Respectively, they correspond to *term*, *sentence*, *clause*, and *goal translations*, respectively. We will denote with $\vec{\mathcal{E}} = (\mathcal{E}_1 \ldots \mathcal{E}_n)$ the transformation specifications that are local to a module, and by $\vec{\mathcal{E}_k}$ the sequence of translations $(\mathcal{E}_{1_k} \ldots \mathcal{E}_{n_k})$, for a particular $k \in \{t, s, c, g\}$.

Figure 3 defines the translations made during the syn-tr phase, started with $\mathbf{tr}[\![\cdot : \mathit{sents}]\!]$. Subscripts will be used to represents families of *kinds*. A term of kind *sents* represents a sequence of sentences, that is translated as a $\mathit{sents}_{\vec{\mathcal{E}_s}}$. The kind sents_{ts} represents the sequences of sentences that require the translation sent_{ts}. The third rule indicates that a sentence of kind $\mathit{sent}_{[t|ts]}$ (we extract the first element of the list of transformations) will be transformed by t, yielding a term of kind sents_{ts} (i.e., a sequence of sentences) on success.[8] In case of failure, the untransformed term will be treated as a sent_{ts}. In this way, all transformations in $\vec{\mathcal{E}_s}$ (i.e., all `sentence_trans`) will be applied. Once ts is empty, the result is translated as kind *term*, equivalent to $\mathit{term}_{\vec{\mathcal{E}_t}}$. Similarly to the previous case, all transformations in ts (i.e., all `term_trans`) are tried and removed from the list of pending transformations. When ts is empty, the datum is treated as an *rterm*, which divides the problem into the translation of arguments as kind *term* and reuniting them as a final (non-suitable for further translations) result. Both transformations are applied in the same order as specified in Ciao.

Figure 4 shows the translations made during the **sem-tr** phase, in this case started with $\mathbf{tr}[\![\cdot : \mathit{clauses}]\!]$. Sequences of clauses are treated in a similar way as for sentences, with the difference that the translation of a clause always gives one clause (not a sequence). When all translations in $\vec{\mathcal{E}_c}$ (all `clause_trans`) have been performed, the head and body are treated. We show no successor for the *head* kind in this figure, since this will be done in the following examples (we could add $\mathit{head} \succ \mathit{final}$ to mark the end of the translation). For *body*, we apply the same body translation on the arguments of control structures (e.g. `,/2`, `;/2`, etc.). If we

[8] We assume that concatenation of sequences is implicit. We can adapt all the discussion to work with lists of sentences, but that would obscure the exposition.

$$
\begin{aligned}
\mathit{clauses} &\succ \mathit{seq}(\mathit{clause}) \\
\mathit{clause} &\succ \mathit{clause}_{\vec{\mathcal{E}_c}} \\
\mathit{clause}_{[t|ts]} &\succ \mathit{try}(t, \mathit{clause}_{ts}, \mathit{clause}_{ts}) \\
\mathit{clause}_{[]} &\succ \mathit{rclause} \\
\textbf{decons}\lfloor \texttt{cl}(h,b) : \mathit{rclause} \rfloor &\rightsquigarrow (h\ b) : (\mathit{head}\ \mathit{body}) \\
\textbf{cons}\lfloor \_ : \mathit{rclause},\ (h\ b) \rfloor &\rightsquigarrow \texttt{cl}(h,b) : \mathit{final}
\end{aligned}
$$

$$
\begin{aligned}
x : \mathit{body} &\Longrightarrow x : \mathit{control} \quad (\text{if } \textsf{term-fn}(x) \in \textsf{control-fn}) \\
\textbf{decons}\lfloor x : \mathit{control} \rfloor &\rightsquigarrow \textsf{get-args}(x) : (\mathit{body} \ldots \mathit{body}) \\
\textbf{cons}\lfloor x : \mathit{control},\ \vec{a} \rfloor &\rightsquigarrow \textsf{set-args}(x, \vec{a}) : \mathit{final} \\
x : \mathit{body} &\Longrightarrow x : \mathit{goal} \quad (\text{if } \textsf{term-fn}(x) \notin \textsf{control-fn}) \\
\mathit{goal} &\succ \mathit{goal}_{\vec{\mathcal{E}_g}} \\
\mathit{goal}_{[t|ts]} &\succ \mathit{try}(t, \mathit{body}, \mathit{goal}_{ts}) \\
\mathit{goal}_{[]} &\succ \mathit{resolv}
\end{aligned}
$$

$$
\textsf{control-fn} = \{\texttt{,/2}, \texttt{;/2}, \texttt{->/2}, \texttt{\textbackslash+/1}, \texttt{!/0}\}
$$

Figure 4: Emulating `clause_trans` and `goal_trans`

are not treating a control structure, the translations in $\vec{\mathcal{E}_g}$ are applied (all `goal_trans`). Note that the first kind in the *try* kind of goals is *goal*. In contrast with other translations, when a goal translation is successfully applied, it is not removed from the list; all translations are applied again. This *fixpoint* semantics was required for the original translation hooks, and has been preserved here (of course, with the same termination problems; tackling those issues is out of the scope of the paper). Similarly to *head*, *resolve* is kept open here, for the same reasons.

Note the flexibility of the base framework. E.g., introducing changes in the expansion rules at fundamental levels can be done, even modularly.

Priority-based Ordering of Transformations Although the transformations in the presented framework establish a fixed ordering, we lose it when considering the sentence, term, clause, and goal translations here. A solution for this issue is the introduction of priorities in each hook, so that all transformations in $\vec{\mathcal{E}}$ can be ordered beforehand. E.g., directives like `:- use_package([dcg, fsyntax])` and `:- use_package([fsyntax, dcg])` are totally equivalent and both apply the transformations in the right order. This represents a very large advantage in practice for users of packages.

4.2 More Advanced Transformations

We show here fragments of two translations that deal with the module system and meta-predicates, and which are not expressible in the old transformation hooks (indeed, they were hardwired in the compiler). We indicate the current module as `cm`. We will assume that

we have access to the information visible during the translation, such as parsed module code, declarations, interfaces, etc.

Example 4.1 (Predicate-based Module System) The following rules perform the module resolution and qualification of all goals in the clauses. Instead of duplicating the logic to locate goal positions, the translations are *inserted* in the right place just after goal expansions are perfomed (Fig. 4).

The translation of *head* replaces the term by one where its symbol has been *module-annotated* with cm (resolved as another symbol that is unique for each module and not directly accesible by user code). The rule for *resolv* does the same, but uses the module obtained from lookup (that indicates where the predicate is defined).[9]

$$
\begin{aligned}
x : \mathit{head} &\Longrightarrow \mathsf{term\text{-}mod\text{-}concat}(\mathsf{cm}, x) : \mathit{final} \\
x : \mathit{resolv} &\Longrightarrow \mathsf{term\text{-}mod\text{-}concat}(m, x') : \mathit{meta} && (\text{if } \mathsf{lookup}(x, m, x')) \\
x : \mathit{resolv} &\Longrightarrow \mathsf{error}(\text{"}\mathit{module\ error}\text{"}) : \mathit{final} && (\text{if } \neg\mathsf{lookup}(a, \_, \_))
\end{aligned}
$$

$$
\mathsf{lookup}(a, m, a') \equiv \begin{cases}
\neg\mathsf{qual}(a, \_, \_) \wedge (\varpi_{\mathsf{cm}} \vdash \mathsf{defined}(fn)) \wedge m = \mathsf{cm} \wedge a' = a \\
\neg\mathsf{qual}(a, \_, \_) \wedge (\varpi_{\mathsf{cm}} \vdash_m \mathsf{exported}(fn)) \wedge a' = a \\
\mathsf{qual}(a, \mathsf{cm}, a') \wedge (\varpi_{\mathsf{cm}} \vdash \mathsf{defined}(fn)) \wedge m = \mathsf{cm} \\
\mathsf{qual}(a, m, a') \wedge m \neq \mathsf{cm} \wedge (\varpi_{\mathsf{cm}} \vdash_m \mathsf{exported}(fn))
\end{cases}
$$

where $fn = \mathsf{term\text{-}fn}(a')$

The complete specification is lengthy, but not more complicated. It requires checking for ambiguity on import (e.g., m in lookup must be unique, etc.). As an example we showed here error reporting as translation to an *error* term. More elabortated error handling can be performed by extending the translation algorithm (e.g., to carry source numbers transparently alongside the terms). □

Example 4.2 (Rules for Meta-predicates) The *meta* translation decomposes the term into meta-arguments, translates them (which gives a pair of the transformed term and an optional goal), and composes back the term by replacing the goal arguments and prefixing the goal with the optional goals from meta-argument expansion. Looking up in the domain of the resolved module the *meta-predicate* properties we get the *meta-type* of each argument in the goal:

$$
\begin{aligned}
\mathbf{decons}\lfloor g : \mathit{meta} \rfloor \rightsquigarrow \vec{x} : \vec{\kappa} \quad & \text{where} \\
& \vec{x} = \mathsf{get\text{-}args}(g) \\
& \varpi_{\mathsf{cm}} \vdash \mathsf{meta\text{-}pred}(\mathsf{term\text{-}fn}(g), \vec{\tau}) \\
& \kappa_i = \mathit{marg}(\tau_i) \quad \forall i. 1 < i < |\vec{\tau}| \\
\mathbf{cons}\lfloor g : \mathit{meta},\ \vec{a} \rfloor \rightsquigarrow g' : \mathit{final} \quad & \text{where} \\
& a_i = (x_i, s_i) \quad \forall i. 1 < i < |\vec{a}| \\
& g' = \mathsf{to\text{-}conj}((s_1\ s_2\ \ldots\ \mathsf{set\text{-}args}(g, \vec{x}))
\end{aligned}
$$

We assume that meta-pred (relating the module-expanded predicate name with its `:- meta_predicate` declaration) has been added to ϖ_{cm}. The to-conj function transforms the input sequence into a conjunction of literals, and ϵ denotes the empty sequence.

[9] $\mathsf{qual}(mg, m, g)$ is true iff the term mg the qualification of term g with term m (e.g., `lists:append([1],[2],[1,2])`). We use it to avoid ambiguity with the colon symbol used elsewhere in rules.

In the expansion of an argument the predicate needs-rt is valid if the term, assuming that it represents a goal, is not known at compile time, that is $\mathsf{needs\text{-}rt}(x) \equiv (v \text{ is variable} \vee (x = qm : \_ \text{ and } qm \text{ is not an atom}))$. In such case the rule emits code that will perform an expansion at run time (which however may share code with those rules).

$$x : \mathit{marg}(\tau) \Longrightarrow (x, \epsilon) : \mathit{final} \qquad (\text{if } \tau \neq \mathsf{goal})$$

$$x : \mathit{marg}(\tau) \Longrightarrow (x', (\mathtt{rtexp}(x, \tau, \mathsf{cm}, x'))) : \mathit{final} \qquad (\text{if } \tau = \mathsf{goal} \wedge \mathsf{needs\text{-}rt}(x))$$

where x' is a new variable

$$\textbf{decons}\lfloor x : \mathit{marg}(\tau) \rfloor \rightsquigarrow x : \mathit{body} \qquad (\text{if } \tau = \mathsf{goal} \wedge \neg\mathsf{needs\text{-}rt}(x))$$

$$\textbf{cons}\lfloor \_ : \mathit{marg}(\tau),\ x \rfloor \rightsquigarrow (x, \epsilon) : \mathit{final}$$

5 Related Work

In addition to the classic examples for imperative languages, such as the `C` preprocessor, or more semantic approaches like `C++` *templates* and Java *generics*, much work has been carried out in the field of extensible syntax and semantics in the context of functional programming. Modern template systems such as the one implemented by the Glasgow Haskell compiler [SJ02] generally provide syntax extensions mechanisms in addition to static metaprogramming. The Objective Caml preprocessor, Camlp4 [dRP] provides similar features but focuses first on the syntax extension aspects. Both systems allow the combination of different syntax within the host language by using explicit mechanisms of quotations/antiquotations.

An other elegant approach consists on defining language extensions based on interpreters. In [Hud98] a methodology for building domain-specific languages is shown, which combines the use of modular monad interpreters with a partial evaluation stage to reduce or eliminate the interpretation overhead. Although this approach provides a clean semantics for the extension, it has the disadvantage of requiring the (not always automatable) partial evaluation phase for efficiency and not being easy to integrate with the rest of the language and with the compilation architecture.

Another solution explored has been to expose the abstract syntax tree, through a reasonable interface, to the extensions. Racket (formerly PLT Scheme) [FP10] has an open macro system providing a flexible mechanism for writing language extensions. It allowed the design of domain-specific languages (including syntax), but also language features such as, e.g., the class and component systems, which in Racket are written using this framework. To the extent of our knowledge, there is no formal description of the framework nor whether and how multiple language extensions interact when specified simultaneously. However, it is interesting to note that despite growing independently, Ciao and Racket, both dynamic languages, have developed similar ideas, like separation of compile-time and run-time affairs and the necessity of expansions at different phases.

Finally, extensibility has also been achieved by making use of rewriting rules. For instance, by mixing such features with compilation inlining, the Glasgow Haskell compiler provides a powerful tool for purely functional code optimization [JTH01]. It seems however the the result of the application of such rules can quickly become unpredictable [EFdM03]. In the context of contraint programming, a successful language transformation tool is Cadmium [DDKS08], which compiles solver-independent constraint models. For this application, the language takes advantage of the distributive property of conjunction to write rules that match against the

conjunctive context.

Although our approach is akin to a rewrite rule system that obtains normalized forms, note that it is easy to combine it with other methods such as, e.g., customized syntactic parsers (beyond defining operators on the fly), more expressive rewrite rules, and reflection of source properties in the abstract syntax tree, e.g., for error reporting or for debugging or analyzing applications (where the original or some prior term is required).

6 Conclusions

We have described an extensible compilation framework for dynamic programming languages that is amenable to performing separate, incremental compilation. Extensibility is ensured by a language of rewrite rules, defined in *pluggable* compilation modules. Although the work is mainly focused on Prolog-like languages, most of the presentation deals with common concepts (modules, interfaces, declarations, identifiers), and thus we believe that it can be adapted to other paradigms with minor effort.

In general, the availability of a rich and expressive extension system is a large advantage for language design. One obvious improvement is that it helps in accommodating the programmers need for syntactic sugar while keeping changes at the kernel language minimum. It also offers benefits for portability, since it makes it possible to keep a common front end (or a set of language features) and *plug in* different kernel engines (e.g., Prolog systems) at the back end, as long as they provide access to the same kernel language (or one that is rich enough) [WSC11].

Beyond the obvious usefulness of the framework as a separation of concerns during the design of extensions, the support for extension composition and separate compilation, etc., the translation rules can also be seen as a complementary specification mechanism for the language features designed. If such rules are succinct and clear enough, which is not that hard in practice, they can actually be exposed to programmers alongside standard documentation. We plan to modify the lpdoc tool [Her00] to provide support for this.

We believe that the model proposed makes it easier to provide unambiguous, composable specifications of language extensions that should not only make reasoning about correctness easier, but also avoid causing and propagating erroneous language design decisions (such as, e.g., unintended compilation dependencies between modules that would ruin any *parallel* compilation or analysis efforts) that are normally hard to detect and correct. We also hope that our contribution will help, in the context of logic programming, set a basis for interoperability and portability of language extensions among different systems.

References

[Aca09] Umut A. Acar. Self-adjusting computation: (an overview). In *PEPM*, pages 1–6, 2009.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *POPL*, pages 266–277, 1997.

[CCH06] A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *The 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, pages 142–162, Fuji Susono (Japan), April 2006.

[CH00] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[DDKS08] Gregory Duck, Leslie De Koninck, and Peter Stuckey. Cadmium: An implementation of acd term rewriting. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 531–545. Springer Berlin / Heidelberg, 2008.

[dRP] Daniel de Rauglaudre and Nicolas Pouillard. Camlp4. `http://brion.inria.fr/gallium/index.php/Camlp4`.

[EFdM03] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003.

[FP10] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[Fw09] Thomas Frühwirth. *Constraint Handling Rules.* Cambridge University Press, August 2009.

[HBC+10] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. Technical Report CLIP2/2010.0, Technical University of Madrid (UPM), School of Computer Science, March 2010. Under consideration for publication in *Theory and Practice of Logic Programming (TPLP).*

[Her00] M. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In *in Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[JTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell workshop*, 2001.

[SJ02] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for haskell. *Haskell workshop*, 37(12):60–75, 2002.

[Wie10] J. Wielemaker. *The SWI-Prolog User's Manual 5.9.9*, 2010. Available from `http://www.swi-prolog.org`.

[WSC11] J. Wielemaker and V. Santos-Costa. On the Portability of Prolog Applications. In *Practical Aspects of Declarative Languages (PADL'11)*, volume 6539 of *LNCS*, pages 69–83. Springer, January 2011.

A Summary of Notation

i, j, k	names for modules (unless used explicitly as sequence indices)
p_i	source code for module i
φ_i	interface of module i
ϖ_i	parsed module i
ω_i	object code for module i
f, g, ...	interpreted (if defined) or uninterpreted function symbol
c, t, ...	names of *kinds* (in translation rules)
$\vec{s}$	sequence with elements $s_1, \ldots s_n$
$\lvert\vec{s}\rvert$	length of sequence $\vec{s}$
ϵ	empty sequence
T	the set of terms (representing the abstract syntax tree)
`foo`, `bar`	concrete symbol names for terms in T (in translation rules)