Modular Extensions for Modular (Logic) Languages

José 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

Abstract. We address the problem of 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 use of the concept of modularity in this context is twofold. We would like these extensions to be modular, in the sense above, 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 Prolog-based systems and languages.

Keywords: Compilation; Modules; Modular Program Processing; Separate Compilation; Prolog; Ciao; Language Extensions; Domain Specific Languages.

1 Introduction

The choice of a good notation and adequate semantics when encoding a particular problem can dramatically affect the final outcome. Extreme examples 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 achieving notational convenience while avoiding inter-language communication costs. In the case of Prolog, language extension through term-expansion systems (combined with operator definitions) has traditionally offered a quick way to develop variants of logic languages and

semantics (e.g., experimental domain-specific languages, constraint systems, optimizations, debugging tools, etc.). Some systems, and in particular Ciao [10], have placed special attention on these capabilities, extending them [1] and exploiting them as the base for many language extensions.

Once a good mechanism is available for writing extensions and a number of them are available, it is natural to consider whether combining a number of them following modular design principles is feasible. For example, consider embedding a simple handy *functional notation* [3] (syntactic sugar to write goals, marked with `~`, in term positions), into a more complex extension, such as the Prolog-based implementation of CHR [8]. In this new dialect, the CHR rule (see Sect. 6.2.1 in Frühwirth's book [8]):

```
T eq and(T1, T2), T1 eq 1, T2 eq X <=> T eq X.
```

can be written more concisely as:

```
T eq and(~eq(1), ~eq(X)) <=> T eq X.
```

Intuitively, expansions are applied one after the other. This already points out that at least a mechanism to determine application order is needed. This is already undesirable because it requires users to be aware of the valid orderings. Furthermore, just ordering may not be enough. In our example, if functional syntax is applied first, it must normalize the `~eq(_)` terms before CHR translation happens, but there is no simple way to indicate to the functional expansion that the CHR constraints have to be treated syntactically as goals. If CHR translation is done first, it will not recognize that `~eq(_)` corresponds to a constraint, and incorrect code will be generated before the functional expansion takes place. Thus, the second rule cannot be translated into the first one by simply composing the two expansions, without tweaking the translation code, which is undesirable.

Moreover, current extension mechanisms have difficulties dealing with the module system. An example is the Typed Prolog extension of [13], which elegantly implements gradually typed Prolog in the style of Hindley-Milner, but needs to treat programs as monolitic, non-modular, units. Even if extensions are made module-aware, the *dynamic* features of traditional Prolog systems present an additional hurdle: programs can change dynamically, and modules may be loaded at runtime, with no clear distinction between program code and translation code, and with no limits on what data is available to the expansion (e.g., consulting the predicates compiled in other arbitrary modules). In the worst case, this leads to a chaotic scenario, where reasoning about language translations is an impossible task.

The previous examples illustrate the limitations of the current extension mechanisms for Prolog and motivate the goals of this work:

– **Predictable Combination of Fine-grained Extensions**: The extension mechanisms must be fine-grained enough to allow rich combinations, but also provide a simple interface for users of extensions. Namely, programmers should be able to write modules using several extensions (e.g., functional notation, DCGs, and profiling), without being required to know the application order of rules or the compatibility of extensions. Obviously, the result of the combination of such extensions must be predictable. That indirectly leads

us to the necessity of describing a precise compilation model that includes compilation and loading of the extension code.
- **Integration with Module Systems**: It is thus necessary to make the extensions module-aware, while at the same time constraining them to respect the module system. For example, it must be possible to determine during expansion to what module a goal being expanded belongs, if that information is available, or to export new declarations. It is well known that modularity, if not designed carefully, can make static analysis impossible [2]. 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. We take as starting point the module and extension system implemented in Ciao [1,10], which is more elaborate than the one offered by traditional Prolog systems. We provide in this paper a refined formal description of the compilation model, and use it to propose and justify the definition of a number of distinct translation phases as well as the information that is available at each one. Then, we propose a set of rule-based extension mechanisms that we argue generalize 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 gives a detailed description of the core translation process for extensions. Section 3 defines a compilation model that integrates the extensions. Section 4 and Section 5 illustrate the rules defined in the previous section by defining several (real-life) language features and extensions. We close with a discussion of related and future work in Section 6, and conclusions in Section 7.

2 Language Extensions as Translation Rules

By language extensions we refer to translations that manipulate a symbolic representation of a given program. For simplicity we will use *terms* representing abstract syntax trees, denoted by $\mathcal{T}$, following the usual definition of *ground* terms in *first order logic*. To simplify notation, we include sequences of terms ($Seq(\mathcal{T})$) as part of $\mathcal{T}$.[3] We also assume some standard definitions and operations on terms: $\mathsf{termFn}(x)$ denotes the $(name, arity)$ of the term, $\mathsf{args} : \mathcal{T} \to Seq(\mathcal{T})$ obtains the term arguments (i.e., the sequence of children), and $\mathsf{setArgs} : \mathcal{T} \times Seq(\mathcal{T}) \to \mathcal{T}$ replaces the arguments of a term.

We use a homogeneous term representation for the program, but terms may represent a variety of language elements. The meaning of each term is often given by its surrounding context. In order to reflect this, each input term is labeled with a symbolic *kind* annotation. That annotation determines which transformation to apply to each term.

We define the main transformation algorithm $\mathbf{tr}[\![x : \kappa]\!] = x'$ in Fig. 1. Given a term x of *kind* κ, it obtains a term x' by applying the available rules. Translation

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

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

Fig. 1. The Transformation Algorithm

ends for a term when the *final* kind is found. The transformation is driven by rules (defined in *compilation modules*). Note that the rules may contain guards in order to make them conditional on the term. Rule $x : \kappa \Longrightarrow x' : \kappa'$ denotes that when a term x of kind κ is found, it is replaced by x' of kind κ'. Rule $\kappa \succ \kappa'$ is the same, but the term is unmodified. Finally, rules $x : \kappa_x \overset{\mathsf{decons}}{\Longrightarrow} \vec{a} : \vec{\kappa}$ and $(x : \kappa_x,\ \vec{a'}) \overset{\mathsf{recons}}{\Longrightarrow} x' : \kappa'_x$ allow the deconstruction (*decons*) of a term into smaller parts, which are translated and then put together by reconstruction of the term (*recons*). Intuitively, this pair of rules allows performing a complex expansion that reuses other rules (which may be defined elsewhere). We will see examples of all these rules later. We divided expansions into finer-grained translations because we want to be able to combine them and to allow them to be interleaved with other rules in such combinations. Monolithic expansions would render their combination infeasible in many cases.

Additionally, there are some rules for *special kinds*, which are provided here for programmer convenience, even if they can be defined in terms of the previous rules. Their meaning is the following: 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 $\mathit{seq}(\kappa)$ kind indicates that the input term is a sequence of elements of kind κ.[4]

Composition of Transformations Note that the transformation algorithm does not make any assumption 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 5 how to give an unambiguous meaning to conflicting rules targeting the same kind.

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

Example 1 (Combining Transformations). Consider the example about merging CHR and *functional syntax* presented in the introduction. It can be solved in our framework by introducing rules such as:

$$(a \setminus b \texttt{ <=> } c) : \mathit{chrclause}_1 \overset{\text{decons}}{\Longrightarrow} (a\ b\ c) : (\mathit{goal}_1\ \mathit{goal}_1\ \mathit{goal}_1)$$
$$(\_ : \mathit{chrclause}_1,\ (a'\ b'\ c')) \overset{\text{recons}}{\Longrightarrow} (a' \setminus b' \texttt{ <=> } c') : \mathit{chrclause}_2$$

Those rules expose the internal structure of some constructs to allow the cooperation between translations. That is, those rules mean that in the middle of the translation from the kinds $\mathit{chrclause}_1$ and $\mathit{chrclause}_2$ we allow treatment of a kind goal_1, which could be treated, e.g., by the functional syntax package. Note that neither the CHR nor the functional package are required to know about the existence of each other.

3 Integration in the Compilation Model

In our compilation model programs are separated into modules. Modules can import and use code from other modules. Additionally, modules may load language extensions through special *compilation modules*. For the sake of simplicity, we will show here the compilation passes required for a simplified language with exported symbols (e.g., predicates) and imported modules. Extending it to support more features is straightforward.

We assume that compilation is performed on a single module at a time, in two phases.[5] Let us also assume that each phase reads and writes a series of initial (sources), temporal, and final compilation results (linkable/executable binaries). We will call those elements *nodes*, since clearly there is a dependency relation between them. In practice, nodes are read and written from a (persistent) memory, that we will abstract here as a mapping V. We denote as $V(n)$ the value of a node. We denote as $V(n) \leftarrow v$ the destructive assignment of v to n in V, and $V(n)$ the value of n in V.

Given a module i, the first phase (strPass) performs the source code (denoted for conciseness as $\mathsf{src}[i]$) parsing and additional processing to obtain the *abstract syntax tree* ($\mathsf{ast}[i]$) and module interface ($\mathsf{itf}[i]$). In order to extend the compilation, we introduce a call to strTr. This will be defined herein in terms of the translation algorithm $\mathbf{tr}[\![\cdot : \tau]\!]$ (Fig. 1), working on the program definitions. We will see actual example definitions for them in Section 5. We call this the *structural* pass, since we can change arbitrarily the structure of the syntax tree, but we are not yet able to fully provide a semantics for the program, which may depend on external definitions from imported modules.[6] Indeed, the information about local definitions (e.g., defined predicates) and the module *interface*

[5] This is common in many languages, since at least two passes are required to allow identifiers in arbitrary order in the program text.

[6] 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.

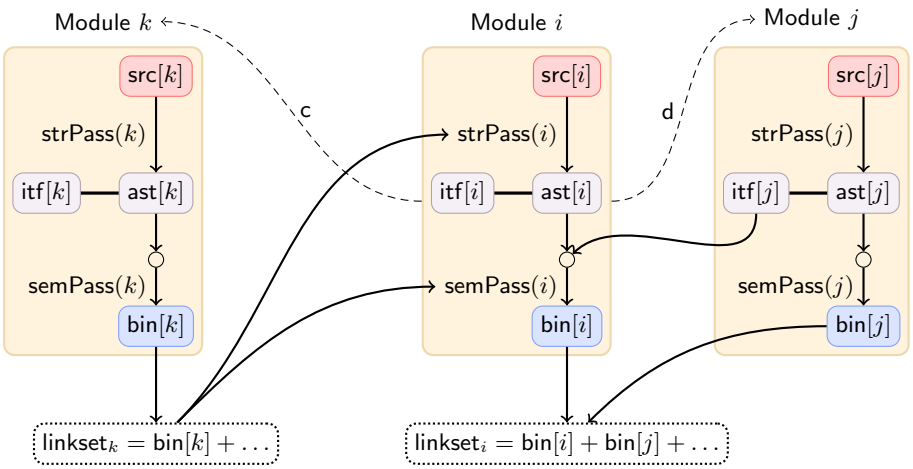

Fig. 2. Example of compilation dependencies for module i, which imports module j (d arrow), and requires compilation module k (c arrow).

(defined below) is available only at the end of this pass. Note that we load compilation modules dynamically during compilation. We will show later how this is done.

Once the first phase finishes, the module interfaces are available. In the second phase (semPass), the interface of the imported modules are collected and processed alongside with the module abstract syntax tree (ast[i]) and interface (itf[i]). The output of this phase (denoted as bin[i]), can be either in an executable form (e.g., bytecode), or in a suitable *kernel* language treatable by other tools (e.g., like program analysis). As in the previous phase, we introduce an extensible translation pass called semTr, similar to strTr. However, this time it can access the interface of imported modules. We name this the *semantic* pass. For loading compilation modules (or any other module) dynamically, we need to compute the *link-set* (the reflexive transitive closure of the *import* relation between modules), or the minimum set of modules required during execution of a module.

Compilation Order In general, determining the order in which compilation must occur, and which recompilations have to take place whenever some source changes is not a straightforward task. For example, see Figure 2 which shows the dependencies for the incremental compilation of a module i depending on module j and compilation module k. We need an algorithm that automatically schedules compilation of modules (both program and translation modules) and which is incremental in order to reduce compilation times. Both of these requirements are necessary in a scalable, useful, dynamic environment. I.e., when developing, the user should not have to do anything special in order to ensure that all mod-

ules are compiled and up to date. However, since dependencies are dynamic we cannot (and would not want to) rely on traditional tools like `Makefile`s.

3.1 Incremental Compilation

```
   data: mappings V, T, and S
 1 def UpdateNode(n):
 2     r = RulingNode(n) ; CheckNode(r)
 3     if S(r) = invalid then
 4         S(r) ← working
 5         if GenNode(r) then S(r) ← valid(T(r)) else S(r) ← error
 6 def CheckNode(r):
 7     if S(r) = ⊥ then
 8         if UpToDate(T(r), NodeInputs(r)) then S(r) ← valid(T(r))
 9         else S(r) ← invalid
10 def UpToDate(t, oldin):
11     if t = ⊥ ∨ oldin = ⊥ then return False
12     foreach n_in ∈ oldin do
13         r_in = RulingNode(n_in) ; CheckNode(r_in)
14         if ¬((S(r_in) = valid(t_in)) ∧ (t_in ≤ t)) then return False
15     return True
```

Algorithm 1: UpdateNode

We solve the problems of determining the compilation order, and making the algorithm incremental, with minor changes. The idea is to invoke the necessary compilation passes before each $V(n)$ is read, in order to access up-to-date values. For that, we define the UpdateNode(n) in Algorithm 1.

The algorithm works with *time-stamps*. We extend the V mapping with an (also persistent) mapping T between nodes and time-stamps, so that: $T(x) = \bot$ if the node does not have any value, and for each $V(n) \leftarrow v$, $T(n)$ is updated with a recent time-stamp. We need another mapping S, that relates a node with its *status*, and is non-persistent and monotonous during a *compilation session* (during which compilation of a set of modules takes place, with no allowed changes in source code). When a *compilation session* starts, we begin with the empty status for all nodes. Finally, we assume that for passes that produce more than one output node (e.g., the interface and the syntax tree), we can choose a fixed one of them as the *ruling node* (e.g., the interface). We denote by RulingNode(n) the ruling node of n.

UpdateNode works by obtaining the ruling node, invoking CheckNode to update its status, and, depending on it, invoking GenNode to (re)generate the outputs if necessary. CheckNode (line 6) examines the node and updates its status.

If the node was visited before, the status will be different from $\bot$, and it will exit. If not, it will check that r is up to date ($\mathsf{UpToDate}(t, oldin)$, line 10) w.r.t. all the dynamic input dependencies ($oldin = \mathsf{NodeInputs}(r)$). In our case, for $\mathsf{strPass}(i)$ the input nodes are $\mathsf{src}[i]$ and the *link-set* of all compilation modules specified in the (old) $\mathsf{itf}[i]$. For $\mathsf{semPass}(i)$ the input nodes are $\mathsf{itf}[i]$ and $\mathsf{itf}[j]$, for each imported module j specified in $\mathsf{itf}[i]$, in addition to the nodes for compilation modules. The input nodes are $\bot$ if it was not possible to obtain them (i.e., no $\mathsf{itf}[i]$ is found). If the node is up-to-date, its status is marked as $\mathsf{valid}(t)$, indicating that it needs no recompilation. If not, it is marked as $\mathsf{invalid}$. This may mark the status of other dependent nodes, but no action is performed on them.

For terminal nodes (e.g., source code $\mathsf{src}[i]$ for some module i), $\mathsf{GenNode}(r)$ will simply check that the node r exists, and $\mathsf{NodeInputs}(r)$ is empty. $\mathsf{CheckNode}$ will mark existing terminal nodes as *valid*. Non-existing nodes will be marked as *invalid*, and later $\mathsf{UpdateNode}$ will try to generate them. Since they do not exist, they will be marked as *error*. For computable nodes, $\mathsf{GenNode}(r)$ invokes the compilation pass that generates the corresponding output ruling node (based on static output dependencies, i.e., $\mathsf{strPass}(i)$ generates $\mathsf{itf}[i]$, $\mathsf{semPass}(i)$ generates $\mathsf{bin}[i]$). If compilation was successful, the status is updated with $\mathsf{valid}(T(r))$ (indicating that it was successfully generated within this *compilation session*). On error, error is used as mark. An additional $\mathsf{working}$ status is used to mark nodes being computed and detect compilation loops (i.e., compilation modules depending on themselves). Note that for nodes whose value is assumed to be always up to date (*frozen nodes*, e.g., precompiled system libraries or static modules that cannot be updated) we make $S(n) = \mathsf{valid}(0)$ by definition (denoting the oldest possible time-stamp).

Correctness and Optimality of Time-stamp Approximation The algorithm is based on, given a node, knowing if it needs to be recomputed. Based on the fact that each compilation pass only depends on its inputs, we can determine this by checking if the contents of a node have changed w.r.t. the contents used for the pass. For that, we could keep all the versions of each node, and number them in increasing order. Instead of local version numbers, we can use *time-stamps*, as a global version counter updated each time a node is written. This has the property that for each generated node n, $T(n) \geq T(m)$ for each m being an input dependency of n. If we can reason on time-stamps, then keeping the contents of each node version is unnecessary.[7] So if we find an input dependency with a time-stamp greater than $T(n)$, then it is possible that it may have changed. We may have false positives (time changed but the value is the same), which will result in more work than necessary, but not incorrect results. If the time-stamp is less or equal then we can be assured that it has not changed since

[7] When dealing with large dependencies, this seems inpractical, both in terms of time and space. We want this operation to be as fast as possible and not consume much additional space.

n was generated. Unless time-stamps are artificially changed by hand, we will not have false negatives (whenever a node needs to be computed, it will be).

We only need to keep for each node its dependencies (the name of the nodes, not their value), or provide a way of inferring them from other stored values.[8]

Handling Compilation Module Loops When a compilation module depends on modules that depend on it, a *deadlock* situation occurs. The compilation module cannot be compiled because it requires some modules that cannot be compiled yet. However, it is common to have languages that compile themselves. We solve the issue by distinguishing between normal and static modules. Static modules have been compiled previously and their $\mathsf{bin}[i]$ and $\mathsf{itf}[i]$ are kept for following compilations (say $\mathsf{bin}[i]^S$ and $\mathsf{itf}[i]^S$ respectively). In that case, $(\mathsf{itf}[i] = \mathsf{itf}[i]^S \wedge \mathsf{bin}[i] = \mathsf{bin}[i]^S)$. 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 (our source, only understood by our compilation module, may be written in a language for which there exists no working compiler).

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 the module interface ($\mathsf{itf}[j]$), which once computed cannot be changed without invalidating the compilation of any module i that imports it j. For this reason, a *semantic* expansion cannot modify the module interface.

4 Backward Compatibility

We now illustrate how the Ciao expansion primitives [1] can be easily emulated within the proposed approach. Ciao extensions are defined in special libraries called *packages*. They contain lexical and syntactic definitions (such as new operators), and hooks for language extension, defined in *compilation modules*. The available hooks can be seen as partial functional relations (or predicates that given an input have at least one solution) that translate different program parts: *term*, *sentence*, *clause*, and *goal translations*. For conciseness, we will denote them as $\mathcal{E}_t$, $\mathcal{E}_s$, $\mathcal{E}_c$, $\mathcal{E}_g \subseteq \mathcal{T} \times \mathcal{T}$, respectively. The transformations in a single package will be the tuple $\mathcal{E} = (\mathcal{E}_t, \mathcal{E}_s, \mathcal{E}_c, \mathcal{E}_g)$. We will denote with $\vec{\mathcal{E}} = (\mathcal{E}_1 \ldots \mathcal{E}_n)$ all the transformation specifications that are local and used in module, and by $\vec{\mathcal{E}_k}$ the sequence of translations $(\mathcal{E}_{1_k} \ldots \mathcal{E}_{n_k})$, for a particular $k \in \{t, s, c, g\}$. Fig. 3 shows the emulation of these translations.

The translations made during the strTr phase start with $\mathbf{tr}[\![\cdot : \mathit{sents}]\!]$. A term of kind sents represents a sequence of sentences, that is translated as a $\mathit{sents}_{\vec{\mathcal{E}_s}}$.

[8] That is of course not necessary for static dependencies (e.g., that each $\mathsf{ast}[i]$ depends on $\mathsf{src}[i]$).

$$
\begin{array}{rl}
\mathit{sents} & \succ \mathit{sents}_{\vec{\mathcal{E}}_s} \\
\mathit{sents}_{ts} & \succ \mathit{seq}(\mathit{sent}_{ts}) \\
\mathit{sent}_{[t|ts]} & \succ \mathit{try}(t, \mathit{sents}_{ts}, \mathit{sent}_{ts}) \\
\mathit{sent}_{[]} & \succ \mathit{term}
\end{array}
\qquad
\begin{array}{rl}
\mathit{term} & \succ \mathit{term}_{\vec{\mathcal{E}}_t} \\
\mathit{term}_{[t|ts]} & \succ \mathit{try}(t, \mathit{term}_{ts}, \mathit{term}_{ts}) \\
\mathit{term}_{[]} & \succ \mathit{rterm} \\
x : \mathit{rterm} & \overset{\text{decons}}{\Longrightarrow} \mathsf{args}(x) : (\mathit{term} \ldots \mathit{term}) \\
(x : \mathit{rterm},\ \vec{a}) & \overset{\text{recons}}{\Longrightarrow} \mathsf{setArgs}(x, \vec{a}) : \mathit{final}
\end{array}
$$

$$
\begin{array}{rl}
\mathit{clauses} & \succ \mathit{seq}(\mathit{clause}) \\
\mathit{clause} & \succ \mathit{clause}_{\vec{\mathcal{E}}_c} \\
\mathit{clause}_{[t|ts]} & \succ \mathit{try}(t, \mathit{clause}_{ts}, \mathit{clause}_{ts}) \\
\mathit{clause}_{[]} & \succ \mathit{hb} \\
(h\,\texttt{:-}\,b) : \mathit{hb} & \overset{\text{decons}}{\Longrightarrow} (h\ b) : (\mathit{head}\ \mathit{body}) \\
(\_ : \mathit{hb},\ (h\ b)) & \overset{\text{recons}}{\Longrightarrow} (h\,\texttt{:-}\,b) : \mathit{final}
\end{array}
\qquad
\begin{array}{rl}
\mathit{body} & \succ \mathit{try}(f, \mathit{control}, \mathit{goal}) \\
f(x, x) & \equiv x \in \{\texttt{,/2}, \texttt{;/2}, \ldots\} \\
x : \mathit{control} & \overset{\text{decons}}{\Longrightarrow} \mathsf{args}(x) : (\mathit{body} \ldots \mathit{body}) \\
(x : \mathit{control},\ \vec{a}) & \overset{\text{recons}}{\Longrightarrow} \mathsf{setArgs}(x, \vec{a}) : \mathit{final} \\
\mathit{goal} & \succ \mathit{goal}_{\vec{\mathcal{E}}_g} \\
\mathit{goal}_{[t|ts]} & \succ \mathit{try}(t, \mathit{body}, \mathit{goal}_{ts}) \\
\mathit{goal}_{[]} & \succ \mathit{resolv}
\end{array}
$$

Fig. 3. Emulating Ciao translation rules

Subscripts are used here to represent families of *kinds*. The kind sents_{ts} represents a sequence 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.[9] 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.

The translations made during the semTr phase start with $\mathbf{tr}[\![\ \cdot\ :\ \mathit{clauses}]\!]$. Sequences of clauses are treated in a similar way as sentences, with the difference that the translation of a clause always returns 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. In this figure, we do not show any successor for the *head* kind, 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 constructs (e.g. `,/2`, `;/2`, etc.). If we are not treating a control structure, the translations in $\vec{\mathcal{E}_g}$ are applied (all `goal_trans`). Note that

[9] 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.

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 tried again until none is applicable.[10] In such case, the term is translated as a *resolve* kind (for the same reason as for *head*, we leave it open for later translations).

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

Priority-based Ordering of Transformations The rules presented in this section establish a precise and fixed application order. However, when more than one sentence, term, clause, or goal translation is used in the same module the ordering among them also needs to be specified. The standard solution for this problem in Ciao is to use the order in which the packages which contain the expansion code are stated (e.g., in `:- use_package([dcg, fsyntax])` the `dcg` transformations precede those of `fsyntax`. We propose an arguably better solution for this problem: to introduce a priority in each hook, so that all transformations in $\vec{\mathcal{E}}$ can be ordered beforehand. With this solution (now implemented in Ciao) directives such as `:- use_package([dcg, fsyntax])` and `:- use_package([fsyntax, dcg])` are fully equivalent, and both would apply the transformations in the right order. Of course, this moves the responsibility from the user of the extension to the extension developer. However, in practice this represents a huge advantage for users of packages.

5 Examples and Applications

We show the expressivity of the rules with fragments of two translations that deal with the module system and meta-predicates. Each of them is presented separately, but their combination results in a transformation equivalent to that hardwired in the Ciao compiler (and which was not expressible in the old transformation hooks). For the sake of clarity, 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.

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 2 (Predicate-based Module System). The following rules perform the module resolution and symbol replacement in all the clause goals to implement a predicate-based module system via a language extension. Instead of duplicating

[10] This preserves the semantics of the original translation hooks, where termination is up to the writer of the translation rules. Detecting those problems is out of the scope of this paper.

the logic to locate goal positions, the translations are *inserted* in the right place just after goal expansions are performed (Fig. 3).

We denote that a predicate symbol f is defined in the current module by $\mathsf{localdef}(f) \equiv \mathsf{defined}(f) \in \mathsf{ast}[\mathsf{cm}]$, and that f is exported by an imported module m by $\mathsf{importdef}(f,m) \equiv (\mathsf{exported}(f) \in \mathsf{itf}[m] \wedge \mathsf{imported}(m) \in \mathsf{ast}[\mathsf{cm}])$. Let $\mathsf{modlocal}(m,t)$ be a term operation that replaces the principal functor of term t by another one that is private to module m (e.g., by a special representation, not directly accessible from user code, that concatenates the name of the current module cm with the symbol). The translation of *head* transforms the term using that operation. The rule for *resolv* does the same, but uses the module obtained from lookup (that indicates where the predicate is defined).[11]

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

$$
\begin{aligned}
\mathsf{lookup}(a, m, a') \equiv\ & (\neg\mathsf{qual}(a, \_, \_) \ \wedge a' = a \wedge \mathsf{localdef}(f) \wedge m = \mathsf{cm}) \vee \\
& (\neg\mathsf{qual}(a, \_, \_) \ \wedge a' = a \wedge \mathsf{importdef}(f, m) \wedge m = \mathsf{cm}) \vee \\
& (\mathsf{qual}(a, m, a') \wedge m = \mathsf{cm} \wedge \mathsf{localdef}(f)) \vee \\
& (\mathsf{qual}(a, m, a') \wedge m \neq \mathsf{cm} \wedge \mathsf{importdef}(f, m)) \\
& \text{where } f = \mathsf{termFn}(a')
\end{aligned}
$$

The complete specification is lengthy, but not more complicated. E.g., it would require more elaborate error handling, which checks for ambiguity on import (e.g., m in lookup must be unique, etc.).

Example 3 (Rules for Meta-predicates). Goals that call meta-predicates in Prolog require special handling of their arguments. We specify the translation of such goals though a kind *meta*. The translation rule decomposes the term into meta-arguments, each of kind $\mathit{marg}(\tau)$, where τ is the meta-type for the predicate, e.g., `goal`). Note that we assume that $\mathsf{ast}[\mathsf{cm}]$ includes a term $\mathsf{metaPred}(f, \vec{\tau})$ for each `:- meta_predicate` declaration. That relates the module-local symbol f of the predicate with each of the *meta-types* of the goal arguments. The translation of $\mathit{marg}(\tau)$ returns a pair of the transformed term and an optional goal. Then, the composition rule rebuilds the goal by placing the transformed terms

[11] $\mathsf{qual}(mg, m, g)$ is true iff the term mg is 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.

as arguments, collecting the optional goals in front of it:

$$
\begin{aligned}
g : \mathit{meta} &\overset{\mathsf{decons}}{\Longrightarrow} \vec{x} : \vec{\kappa} \quad \text{where} \\
&\quad \vec{x} = \mathsf{args}(g) \\
&\quad \mathsf{metaPred}(\mathsf{termFn}(g), \vec{\tau}) \in \mathsf{ast}[\mathsf{cm}] \\
&\quad \kappa_i = \mathit{marg}(\tau_i) \quad \forall i.1 < i < |\vec{\tau}| \\
(g : \mathit{meta},\ \vec{a}) &\overset{\mathsf{recons}}{\Longrightarrow} g' : \mathit{final} \quad \text{where} \\
&\quad a_i = (x_i, s_i) \quad \forall i.1 < i < l, \quad l = |\vec{a}|, \\
&\quad g' = \mathsf{toConj}((s_1\ s_2\ \ldots\ s_l\ \mathsf{setArgs}(g, \vec{x})))
\end{aligned}
$$

The toConj function transforms the input sequence into a conjunction of literals. We list below the rules for arguments. In the cases where the arguments do not need any treatment, we use ϵ as the second element in the pair, which denotes the empty sequence. The case where the argument represents a goal, but is not known at compile time (e.g., x is a variable, or $x = qm : \_$, where qm is not an atom), is captured by $\mathsf{needsRt}(x)$. In such case the rule emits code that will perform an expansion at run time (which however may share code with those rules). Finally, if the argument represents a goal, we use a deconstruction rule to expose an argument of kind *body*, which once translated is put back in a pair, as required by $\mathit{marg}(\_)$.[12]

$$
\begin{aligned}
x : \mathit{marg}(\tau) &\Longrightarrow (x, \epsilon) : \mathit{final} && (\text{if } \tau \neq \mathsf{goal}) \\
x : \mathit{marg}(\tau) &\Longrightarrow (x', (\mathtt{rtexp}(x, \tau, \mathsf{cm}, x'))) : \mathit{final} && (\text{if } \tau = \mathsf{goal} \wedge \mathsf{needsRt}(x)) \\
&\quad \text{where } x' \text{ is a new variable} \\
x : \mathit{marg}(\tau) &\overset{\mathsf{decons}}{\Longrightarrow} x : \mathit{body} && (\text{if } \tau = \mathsf{goal} \wedge \neg\mathsf{needsRt}(x)) \\
(\_ : \mathit{marg}(\tau),\ x) &\overset{\mathsf{recons}}{\Longrightarrow} (x, \epsilon) : \mathit{final}
\end{aligned}
$$

Example 4 (Combined Transformation). The previous transformations can be combined to translate goals involving meta-predicate calls into plain module-qualified goals. The rules defined in this section and in Section 4 can be used to transform the input goal:

$$G = \texttt{findall(X,member(X,[1,2,3]),Xs)}$$

as G' by evaluating $\mathbf{tr}[\![G : \mathit{goal}]\!]$ so that:

$$G' = \texttt{'aggregates:findall'(X,'lists:member'(X,[1,2,3]),Xs)}$$

assuming that `findall/3` is imported from the module `aggregates`, that `member/2` is imported from `lists`, and that the meta-predicate declaration of `findall/3` specifies that its second argument is a goal.

[12] This allows applying rules treating *bodies*, such as symbol renaming for the module system.

6 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 in the Glasgow Haskell compiler [14] generally provide syntax extension mechanisms in addition to static metaprogramming. The Objective Caml preprocessor, Camlp4 [4], provides similar features but focuses more on the syntax extension aspects. Both systems allow the combination of different syntax within the host language by using explicit mechanisms of quotations/antiquotations.

Another elegant approach consists on defining language extensions based on interpreters. In [11] 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 its integration with the rest of the language and with the compilation architecture is more complex.

Another solution explored has been to expose the abstract syntax tree, through a reasonable interface, to the extensions. Racket (formerly PLT Scheme) [7] 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 [12]. It seems however that the result of the application of such rules can quickly become unpredictable [6]. In the context of constraint programming, a successful language transformation tool is Cadmium [5], which compiles solver-independent constraint models.

7 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, inter-

faces, 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 asset for language design. One obvious advantage is that it helps accommodate the programmer's need for syntactic sugar, while keeping changes in the kernel language at a 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) [15].

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 [9] 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 contribute, in the context of logic programming, towards setting a basis for interoperability and portability of language extensions among different systems.

Acknowledgments: This work was funded in part by EU project IST-215483 *SCUBE*, MICINN project TIN-2008-05624 *DOVES*, and CAM project S2009TIC-1465 *PROMETIDOS*.

References

1. Cabeza, D., Hermenegildo, M.: A New Module System for Prolog. In: International Conference on Computational Logic, CL2000. pp. 131–148. No. 1861 in LNAI, Springer-Verlag (July 2000)
2. Cardelli, L.: Program fragments, linking, and modularization. In: POPL. pp. 266–277 (1997)
3. Casas, A., Cabeza, D., Hermenegildo, M.: 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). pp. 142–162. Fuji Susono (Japan) (April 2006)
4. de Rauglaudre, D., Pouillard, N.: Camlp4, http://brion.inria.fr/gallium/index.php/Camlp4
5. Duck, G., De Koninck, L., Stuckey, P.: Cadmium: An implementation of acd term rewriting. In: Garcia de la Banda, M., Pontelli, E. (eds.) Logic Programming, Lecture Notes in Computer Science, vol. 5366, pp. 531–545. Springer Berlin / Heidelberg (2008)

6. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. J. Funct. Program. 13(3), 455–481 (2003)
7. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. (2010), http://racket-lang.org/tr1/
8. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (August 2009)
9. Hermenegildo, M.: A Documentation Generator for (C)LP Systems. In: International Conference on Computational Logic, CL2000. pp. 1345–1361. No. 1861 in LNAI, Springer-Verlag (July 2000)
10. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming (2011), http://arxiv.org/abs/1102.5497
11. Hudak, P.: Modular domain specific languages and tools. In: in Proceedings of Fifth International Conference on Software Reuse. pp. 134–142. IEEE Computer Society Press (1998)
12. Jones, S.P., Tolmach, A., Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in ghc. In: Haskell workshop (2001)
13. Schrijvers, T., Costa, V.S., Wielemaker, J., Demoen, B.: Towards Typed Prolog. In: Pontelli, E., de la Banda, M.M.G. (eds.) International Conference on Logic Programming. pp. 693–697. No. 5366 in LNCS, Springer Verlag (December 2008)
14. Sheard, T., Jones, S.L.P.: Template meta-programming for haskell. Haskell workshop 37(12), 60–75 (2002)
15. Wielemaker, J., Santos-Costa, V.: On the Portability of Prolog Applications. In: Practical Aspects of Declarative Languages (PADL'11). LNCS, vol. 6539, pp. 69–83. Springer (January 2011)