Reversible Language Extensions and their Application in Debugging

Zoé Drey[1], José F. Morales[1], Manuel V. Hermenegildo[1,2], and Manuel Carro[1,2]

[1] IMDEA Software Institute, Madrid (Spain)
[2] School of Computer Science, T. U. Madrid (UPM), (Spain)

Abstract. A range of methodologies and techniques are available to guide the design and implementation of language extensions and domain-specific languages on top of a base language. A simple yet powerful technique to this end is to formulate the extension via source-to-source transformation rules that are interleaved across the different compilation passes of the base language. Despite being a very successful approach, it has the main drawback that the input source code is lost in the process. As a result, during the whole workflow of program development (warning and error reporting, source-level debugging, or even program analysis) the tools involved report in terms of the base language, which is confusing to users. In this paper, we propose an augmented approach to language extensions for Prolog, where symbolic annotations are included in the target program. These annotations allow the selective reversal of the translated code. We illustrate the approach by showing that coupling it with minimal extensions to a generic Prolog debugger allows us to provide users with a familiar, source-level view during the debugging of programs which use a variety of language extensions, such as functional notation, DCGs, or CLP{Q,R}.

Keywords: language extensions, debuggers, logic programming, constraint programming

1 Introduction

One of the key decisions when specifying a problem or writing a program to solve it is choosing the right language. Even when using recent high-level and multi-paradigm languages, the programmer often still needs precise, domain-specific vocabulary, notations, and abstractions which are usually not readily available. These needs are the main motivation behind the development of domain-specific languages, which enable domain experts to express their solutions in terms of the most appropriate constructs.

However, designing a new language can be an intimidating task. A range of methodologies and tools have been developed over the years in order to simplify this process, from compiler-compilers to visual environments [13]. A simple, yet powerful technique for the implementation of domain-specific languages is based on source-to-source transformations. Although in this process the source and target language can be completely different, it is frequent to be just interested in some *idiomatic extensions*, *i.e.,* adding domain-specific features to a host language while preserving the availability of most of the facilities of this language. Examples of such extensions are adding functional notation to a language that does not support it, adding a special notation for grammars (such as Definite Clause Grammars (DCGs) [16]), *etc.* Such transformations have been proposed in the context of object-oriented programming (*e.g.,* Polyglot for Java, [15]), functional programming (*e.g.,* Embedded DSL facilities for Haskell, [9]), or logic programming (*e.g.,* the term_expansion facility in most Prolog systems, or the extended mechanisms of Ciao [2,8]). In this approach, the language implementations provide a collection of *hooks* that allow the programmer to extend the compiler and implement both syntactic and semantic variations.

An important practical aspect is that, in addition to appropriate notation, the programmer also needs environments that help during program development. In particular, basic tools such as editors, analyzers, and, specially, debuggers are fundamental to productivity. However, in contrast to the significant attention given to mechanisms and tools for defining language extensions, comparatively few approaches have been proposed for the efficient construction of such development environments for domain-specific languages. In some cases ad-hoc editors, debuggers, analyzers, *etc.* have been developed from scratch. However, this approach is time consuming, error prone, hard to maintain, and usually not scalable to a variety of language extensions.

A more attractive alternative, at least conceptually, is to reuse the tools available for the target language, such as its debuggers or analyzers. This can in principle save much implementation effort, in the same way in which the source-to-source approach leverages the implementation of the target language to support the domain-specific extensions. However, the downside of this approach is that these tools will obviously communicate with the programmer in terms of the target language. Since a good part of the syntactic structure of the input source code is lost in the transformation process, these messages and debugger steps in terms of the target language are often not easy to relate with the source level and then the target language tools are not really useful for their intended purposes. For example, a debugging trace may display auxiliary calls, temporary variables, and obscure data encodings, with no trivial relation with the control or data domain at the source level. Much of that information is not only hard to read, but in most cases it should be invisible to the programmer or domain expert, who should not be forced to understand how the language at the source level is embedded in the supporting language.

In this paper, we propose an approach for recovering *symbolically* the source of particular translations (that is, *reversing* them and providing an *unexpanded* view when required) in order to make target language level development tools useful in the presence of language extensions. Our solution is presented in the context of Ciao [8], which uses a powerful language extension mechanism for supporting several paradigms and (sub-)languages. We augment this extension mechanism with support for symbolic annotations that enable the recovery of the source code information at the target level. As an example application, we use these annotations to parameterize the Ciao interactive debugger, so that it displays domain-specific information, instead of plain Prolog goals. Our approach requires only very small modifications in the debugger and the compiler, which can still handle other language extensions in the usual way.

The paper is organized as follows: Section 2 presents a concrete extension mechanism and illustrates the limitations of the traditional translation approach in our context. Section 3 presents our approach to unexpansion, and guidelines for instrumenting language extensions so that the intervening translations can be reversed as needed into their input source code. Section 4 presents the application of the approach to the case of debuggers. Finally, Section 5 presents related work and Section 6 concludes and suggests some future work.

2 Language extensions and their limitations

We present a concrete language extension mechanism based on translations (the one implemented in the Ciao language) and then illustrate the limitations of the traditional translation-based extension approach in our context. In Ciao [8], language extensions are implemented through *packages* [2], which encapsulate syntactic extensions for the input language, translation rules for code generation to support new semantics, and the necessary run-time code. Packages are separated into compile-time and run-time parts. The compile-time parts (termed *compilation modules*) are only invoked during compilation, and are not included in executables, since they are not necessary during execution. On the other hand, the run-time parts are only required for execution and are consequently included in executables. This phase distinction has a number of practical advantages, including obviously the reduction of executable sizes.

More formally, let us assume that an extension for some language denoted as $\mathcal{L}_e$ is defined by package PkgMod_e, and that the compiler passes include calls to a generic expansion mechanism $[\![\mathit{expand}]\!]$, which takes a package, an input program in the source language, and generates a program in the target language $\mathcal{L}$. That is, given $[\![\mathit{expand}]\!]_e = [\![\mathit{expand}]\!](\mathit{PkgMod}_e)$, for a program $P_e \in \mathcal{L}_e$ we can obtain the expanded version $[\![\mathit{expand}]\!]_e(P_e) = P \in \mathcal{L}$. Note that in practice, Ciao contains finely grained translation hooks, which allow a better integration with the module system and the composition of translations [14]. This level of detail is not necessary for the scope of this paper; thus, for the sake of simplicity, the expansion will work on whole programs at a time.

Functional notation. We illustrate the translation process in Ciao with an example from the *functional notation* package [3]. This package extends the language with *functional*-like syntax for relations. Informally, this extension allows including terms with predicate symbols as part of data terms, while interpreting them as predicate calls *with an implicit last argument*. It also allows defining clauses in functional style where the last argument is separated by a `:=` symbol (other functionalities are provided, such as expanding goals in the last argument after the body). The translation can be abstractly specified as a collection of rewrite rules such as:

$$\begin{array}{rl}\text{(Clauses)} & \textbf{tr}[\![\ p(\bar{a})\ \texttt{:=}\ C\ \texttt{:-}\ B\ ]\!] = (p'(\bar{v},T)\ \texttt{:-}\ \bar{v}\ \texttt{=}\ \bar{a}, B, T\ \texttt{=}\ C) \\ \text{(Calls)} & \textbf{tr}[\![\ q(\ldots p(\bar{a})\ldots)\ ]\!] = (p'(\bar{a},T), q(\ldots T\ldots))\end{array}$$

The first rule describes the meaning of a clause in functional notation, where p' is the predicate in plain syntax corresponding to the definition of p in functional notation (*i.e.,* using `:=`). The second rule must be applied using a leftmost-innermost strategy for every p function symbol that appears in the goal q, where T is a new variable (skipping higher-order terms).

The usual evaluation order in logic programming corresponds to eager, call-by-value evaluation (but lazy evaluation is possible as shown in [3]). We refer to the actual implementation later in this section.

Example 1. The program excerpt below defines a predicate `f/2` in functional notation and its translation into plain Prolog code. Its body contains nested calls to `k/2` and `l/2`, and also syntactic sugar for a conditional (if-then-else) construct (using the syntax: *CondGoal* ? *ThenExpr* | *ElseExpr*) .

Source code (functional notation)

```
f(X) := X < 42 ?
          (k(l(m(X))) * 3)
        | 1000.
k(X) := X + 1.
l(X) := X - 2.
m(X) := X.
```

Target code (plain Prolog)

```
f(X,Res) :- X < 42, !,
    m(X, M), l(M, L), k(L, K),
    T is K * 3, T = Res.
f(X,1000).
k(X,Res) :- Res is X+1.
l(X,Res) :- Res is X-2.
m(X, X).
```

Forgetful translations and loss of symbolic information. Both the standard compilation and the translations for language extensions are typically focused on implementing some precise semantics during execution. That is, the correctness of the translation guarantees that for all programs $P_e \in \mathcal{L}_e$, the expected semantics $[\![exec]\!]_e$ for that language can be described in terms of a program $P \in \mathcal{L}$ and its corresponding execution mechanism $[\![exec]\!]$. That is, for all $P_e \in \mathcal{L}_e$ there exists a $P = [\![expand]\!]_e(P_e)$ so that $[\![exec]\!]_e(P_e) = [\![exec]\!](P)$.

Most of the time, symbolic information at the source level is lost, since it is not necessary at run time. In particular, such information removal and loss of structure is necessary to perform important program optimizations (*e.g.,* assigning some variables to registers without needing to keep the symbolic name, its relation to other variables in the same scope, *etc.*). When programs are not executed, but manipulated at a symbolic level, the translation-based approach is no longer valid on its own. For example, assume a simple *debugger* that interprets the source and allows the user to inspect variable values at each program point interactively. In this case the translation, as a program transformation, must

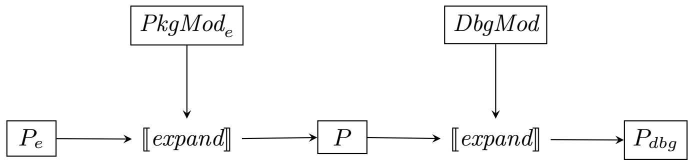

Fig. 1. The translation process and application of the standard debugger.

```
2 2 Call: f(3,_6378) ?
3 3 Call: <(3,42) ?
4 3 Call: m(3,_6658) ?
5 3 Call: l(3,_6663) ?
6 4 Call: is(_6663,3-2) ?
...
9 3 Call: is(_6673,2*3) ?
10 3 Call: =(_6378,6) ?
```

Fig. 2. Excerpt of the display of the interactive debugger.

preserve not only the input/output behaviour but also some other *observable* features (such as line numbers or variable names).

In order to explore the particular case of debuggers more closely, Figure 1 illustrates the translation process of a source program, using a compilation module $PkgMod_e$ containing the translation rules for extension e. If the developer asks the Ciao interpreter to debug this program, further instrumentation is applied that is also defined in part as a language extension, *DbgMod* in Figure 1; this instrumentation customizes the code by encapsulating it into a predicate that specifies whether a part of the code is *spy-able* or not. The following example illustrates in a concrete case the limitations of this process.

Example 2 (Interactive debugging). Consider the code and transformation of Example 1. If the target-level debugger is used without any other provision, following the process of Figure 1, debugging a call to `f(3,T)` amounts to debugging its translation, as illustrated in the trace of Figure 2 (the exit calls are omitted in order to save space). The problem of this trace is twofold: first, the interactive debugging does not make explicit the actual source-level predicate that is currently being tested. Second, understanding the trace forces the developer to make the mental effort of analyzing the debugged data and mapping it back to the source code. This effort increases if the source code contains operators that do not exist on the target (Prolog) side. The first case can be easily overcome when operator definitions are shared, *e.g.,* using a graphical editor and catching the operator with the line number and the occurence number of the call. However, the second case implies remembering the mapping between the source and the target operator. Furthermore, things get even more tedious when one instruction in the source language is translated into a composition of goals.

$$
\begin{tikzcd}
P_e \arrow[r, "{[\![expand]\!]_e}"'] \arrow[d, "\mathsf{Obs}_e(i)"'] & P \arrow[d, "\mathsf{Obs}(i)"'] \\
V_e & V
\end{tikzcd}
\qquad
\begin{tikzcd}
P_e \arrow[r, "{[\![expand]\!]_e^{sym}}"'] \arrow[d, "\mathsf{Obs}_e(i)"'] & (P, Sym) \arrow[d, "\mathsf{Obs}^{sym}(i)"] \arrow[dl, dashed, "\mathsf{Obs}_e^{sym}(i)"] \\
V_e & V \arrow[l, "unexp_v"']
\end{tikzcd}
$$

Fig. 3. Observation problem at the source level (left); Observation using symbolic information (right).

3 Building reversible extensions

In this section we provide an informal definition of *unexpansion* with respect to a language extension. We then present guidelines in order to instrument a compilation module for such a language extension. This instrumentation aims to drive the process of reconstructing a program in terms of the language extension (or *source* language) in which the program is written. Through this mechanism, a language extension can be made *reversible*. To illustrate our purpose, we apply the guidelines and parameterize one of the translation rules used in the functional notation extension.

3.1 Expansion, unexpansion, and observers

We use the term *unexpansion* to designate the inverse of the expansion $[\![expand]\!]_e$, that is, the recovering of the original P_e source program from P. Unfortunately, this inverse is rarely a one-to-one mapping. For example, `f(3,T)` in $\mathcal{L}$ corresponds to both `T=f(3)` and `f(3,T)` (with `f/1` using functional notation). For another example, a clause can either be translated into one or many clauses, as depicted in Example 1 for `f` in functional notation.

Having multiple solutions for unexpansion can be confusing for the user and impractical for automatic transformations. However, the most important use of unexpansion in our context is to observe the behavior of only certain program aspects at the source language level. In this case, unexpansion seems more treatable. For that purpose we define the term *observer* accordingly: an *observer* is an interface that provides some specific source-level information about a particular program. The observer can be either static or dynamic. Specifically, we can consider as observers monitors (*e.g.,* interactive debuggers, tracers, and profilers) for dynamic observation, and verifiers (*e.g.,* static analyzers and model checkers) for static observation. Thus, a source-level view may correspond to the current instruction being invoked in an interactive debugger, or to a trace of the memory state, in a tracer, or perhaps the dependencies between the program variables, in a static analyzer, all of them represented in terms of the source language abstractions.

The correspondance between expansion and unexpansion, in the context of an observer, is sketched in Figure 3. We assume that we have observers $\mathsf{Obs}_e(i)$

and $\mathsf{Obs}(i)$ for the source and target languages, respectively. We denote by i some particular observable aspect and by V the aspect (*e.g.,* "line numbers" and an integer). On the left diagram we depict the impossibility of getting information at the $\mathcal{L}_e$ level in general. To provide the programmer with source-level observers, our approach relies on extending the expansion ($[\![expand]\!]_e^{sym}$) with additional symbolic information (which can be significantly smaller than the sources). Then, observers $\mathsf{Obs}^{sym}(i)$ can retrieve V (*e.g.,* a single number encoding the row and columns) and map it back to V_e (*e.g.,* the row and columns). This composition provides an effective $\mathsf{Obs}_e^{sym}(i)$.

We now propose guidelines for easily instrumenting the translation module of a language extension, in such a way that observers can be parameterized with respect to this instrumentation.

3.2 Instrumentation of a compilation module

Instrumenting a compilation module involves annotating its translation rules with source code information that can then be used by an observer (*i.e.,* the debugger in our application example). We illustrate the instrumentation process on the functional extension example.

Guidelines. The first step in making a language extension reversible is to determine which parts of the source code need to be kept available in the expansion process. The second step is to determine how and where to propagate this information, so that it can be accessed whenever the developer requires observation during program execution. The third step is to determine the representation of the observable data.

Event and data analysis. What events do we want to observe? What do we want to observe about them? These selections should be useful for following the control flow and state changes during program execution. For example, in a λ-calculus-like language, the definition and the application of a function are two of the key elements to follow in order to debug a program [17]. As another example, in a goal involving expressions in functional notation, the debugger must be aware of which positions correspond to data terms and which positions to predicate calls.

Decomposition. How is a source statement decomposed into target code? The answer to this question implies in part how the data that we want to observe should be propagated. For example, while the generic debugger may step through a number of target-level statements, a source-specific debugger may have to consider a single source statement as corresponding to all those steps. This applies for example in the conditional statement `C ? A | B` of the functional notation, where `A` is translated into an (at least) two-goal target code segment.

Representation. How should the data to be observed be represented? In a purely syntactic extension, data always represents elements of the concrete syntax. Nevertheless, it is interesting to consider this question when displaying the runtime

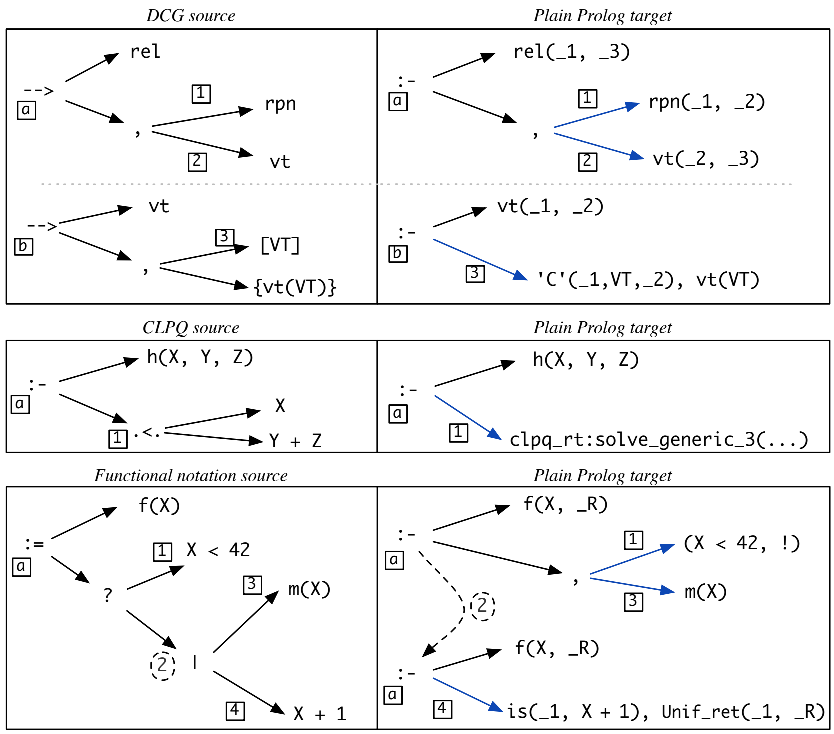

Fig. 4. Annotated translation: source-level code (left) and target-level code (right).

context, such as the state of the memory, for semantic extensions. For example, in a CLP{Q,R} extension, variables are bound at run-time to complex terms attached to attributed variables which reflect the internal, low-level representation of the constraint store, while what the programmer would like to see is a symbolic representation of the constraints among the variables in the source constraint language.

Some examples. We now present three excerpts of code written using language extensions, namely CLPQ (the case for CLPR is analogous), DCG, and the functional notation. The source-level code is represented as a concrete syntax tree in Figure 4, left part. It is translated into Prolog code as depicted in Figure 4, right part.

The annotation process implies tagging each element of the source code that is intended to be observed, *i.e.,* each element meant to be or to refer to a “first-class” concept of the language extension, whether it is defined as a clause or a goal. For example, in a DCG rule such as “`rel --> rpn, vt`” (Figure 4, top part), `rpn` and `vt` are annotated, as these functors lead to a call to another grammar rule. In contrast, elements surrounded by brackets (*e.g.,* `{vt(VT)}`) correspond to plain Prolog, and thus do not require any source level-related special handling. During the expansion process of the source-level code, the target goals correspondingly keep a reference to the source identifiers.

3.3 Implementation of the instrumentation process

To instrument the translation rules we propose to annotate the target parameter of each rule (*i.e.,* the argument in which the code generated by the translation is returned). This annotation is defined as a predicate and provides the symbolic information, encoded as a Prolog term, to drive the process of recovering source code data within the observer. Symbolic information could be a list of variables (along with a function that recovers their value at the source level from the target context, *i.e.,* its environment and store), or a single string to be displayed at the observer output at run time, *e.g.,* in the case of an interactive debugger.

Definitions. We currently distinguish two types of annotations: the `$clause_info` annotation, which is wrapped around target clauses, and the `$goal_info` annotation (or meta-information), which is wrapped around target goals. The purpose of each of these annotations is to gather symbolic information to recover a source-level statement or a source-level call, respectively. Additionally, this distinction enables handling clauses and goals properly, in particular to retrieve their location in source modules.

Both annotations are handled according to the wrapped target element. Specifically, `$clause_info` takes three arguments: the target clauses (`Clauses` in Example 3), a source-level representation (`SI`), and an identifier (`Id`) to optionally enable later retrieval of the translated statement. The body of the source-level statement is itself tagged so as to map to the corresponding target goals when the statement is evaluated. The `$goal_info` annotation takes two arguments: the target goal or goal composition, and an identifier, enabling the retrieval of the source-level call in the body of a statement.

Application to the functional notation translation. We illustrate this annotation process with Example 3. Specifically, we instrument the two translation predicates `defunc` and `defunc_goal`, translating the source-level clauses and goals. The text in italic corresponds to the instrumentation code added over the original translation predicates. Note that defining this code's body (according to the data to observe) can defined as a hook of an extension module.

Example 3. Instrumentation of the translation rules for functional notation.

```
defunc((FuncHead := FuncVal), $clause_info(Clauses, Id, SI) :-
    identify_functional_calls(FuncVal, FuncVal_withIds),
    defunc_rec((FuncHead := FuncVal_withIds), Clauses, SI0),
    build_syminfo([FuncHead,':='|SI0],SI).

defunc_rec((FuncHead := FuncValOpts_withIds), Clauses, [SI1|SIR]) :-
    FuncValOpts_withIds = (FuncVal1 | FuncValR), !,
    Clauses = [Clause1 | ClauseR],
```

```
    defunc_rec((FuncHead := FuncVal1), Clause1, SI1), (1)
    defunc_rec((FuncHead := FuncValR), ClauseR, SIR). (2)

defunc_goal(FuncCall, Goal) :-
    recover_id(FuncCall, Id),
    normalize(FuncCall, NormGoal),...,
    Goal = 'Eval'('$goal_info'(NormGoal, Id), _Ret_Arg).
```

Clause translation The `FuncHead` part on the left corresponds to a predicate declaration; the `FuncValOpts` part on the right corresponds to goal invocations (this results from the data analysis guideline). As introduced in Section 3.2, the source-level elements of the predicate body are tagged using the user-defined functor `identify_functional_calls`. Specifically, each relevant element of the body is associated with a unique identifier, in order to enable the retrieval of its position by the observer.

The symbolic information attached to the annotation is represented by the contents of variable `SI`, created by predicate `build_syminfo`. This variable is handled by an observer, according to the nature of the program view it aims to provide. For example, line numbers, variables, or function names can be attached to it. It can even be left as a free variable, if the observer can automatically retrieve the information.

Notice that the declaration `FuncHead := FuncValOpts` is decomposed into many goals, marked *(1)* and *(2)*, if the `|` operator appears inside its right part. Therefore, the translation needs to indicate to the observers that the declaration is to be treated as a single one. This is done by grouping the symbolic information computed by the evaluation of goals *(1)* and *(2)* into the `$clause_info` wrapper, set as its last argument in the first predicate `defunc`.

Goal translation To relate target goals with their symbolic information the goal translation predicates are instrumented with (1) predicate `recover_id` which extracts identifier `Id` of the annotated source-level goal, and (2) predicate `$goal_info` which is wrapped around the translation code `NormGoal`. `Eval` is not wrapped since it is an intermediate goal in the expansion process.

This approach based on symbolic information enables us to envision a range of program views, from simple syntax recovery to high-level representation of analysis results: annotations can be enriched with source-specific procedures to handle various representations of the target program, enabling different instantiations of the annotation variable. They can even hold procedures that perform advanced computations parameterized with the symbolic information (*e.g.,* counting the number of times a function is invoked).

The instrumentation method is outlined in the schema of Figure 5, which depicts a declaration of the form `f(X) :=` *Cond* `?` B_1 `|` B_2. In this figure, the variable names Sx correspond to identifiers of some program elements associated with some symbolic information, and the expressions **tr**$[\![x]\!]$ correspond to a translation of the term x. Overlined elements represent syntactic nodes.

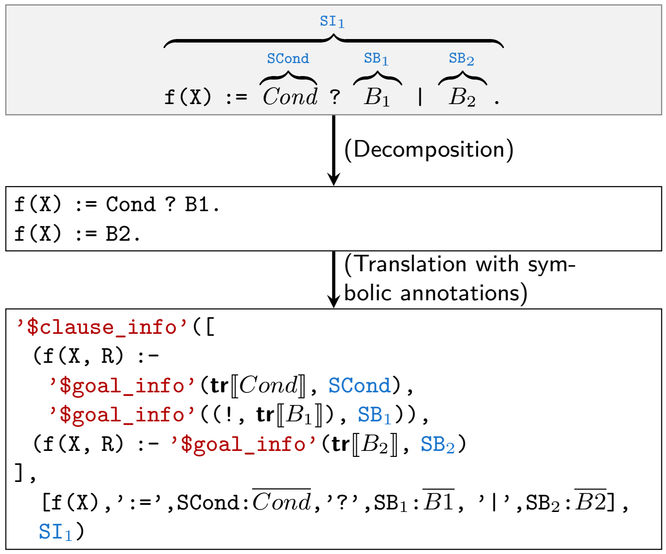

Fig. 5. Instrumented translation of a clause in functional notation.

Implementation overview. The annotations are integrated into the compilation process of an extension module. The overall process of making program behavior observable at the source level through an observer is depicted in Figure 6. Let us describe the extractor and the controller parts of this process.

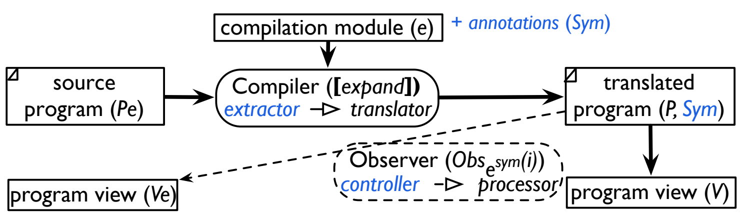

Fig. 6. Overview of the compilation process extended with annotations.

Extending the translation process: extractor. Handling the annotations required a slight extension of the translation phase of the compiler. The extension is as follows: the `$clause_info` annotation is handled inside the original processing step of clauses; the `$goal_info` annotation is handled in the processing step of goals. Specifically, the annotation attached to each clause is encapsulated into an auxiliary clause generated in the same phase as the regular target clause(s). In doing so, the symbolic information is available whenever an observer needs it. During the goal translation step, the target goals that are wrapped by a `$goal_info` annotation are kept executable, enabling the annotation to be ignored whenever the source code needs to be processed by a target-level observer.

Customizing the observer: controller Existing observers for Prolog (such as debuggers, profilers, or static analyzers) can then be customized according to the generated symbolic information. This customizing process is necessarily application-specific, but we provide some general hints as guidelines:

(Hint 1) Collect all the elements requested by the observer (*e.g.,* line numbers, function names, or some text) and make them accessible as a data structure in the arguments of the observer predicates.

(Hint 2) Complement the observer functions that do some processing on some element (clause or goal), by checking if this element holds some symbolic information (*i.e.,* as `$clause_info` or `$goal_info`). For example, if a `$goal_info` wrapper is encountered, extract the name of the source-level goal and make it display it instead of the name of its corresponding target-level goal(s). Otherwise, preserve the target-level behavior.

(Hint 3) Identify locations in the observer where it needs to remember the last data computed with source-level information. For example, such data can correspond to a counter of execution times of a given function, when the observer is a profiler.

4 Application to the interactive debugger

We now illustrate the use of a reversible language extension to parameterize the generic interactive debugger of Ciao. We describe the modifications performed on the debugger, and show the resulting source-level trace for our initial example (Example 1).

4.1 Implementation details

In the case of the debugger, the required symbolic information corresponds to a source node (*e.g.,* `[k(X), ':=', +(X, 1)]` as in Example 1). As a result, the extraction process consists solely of storing each source node before its expansion.

Once the source-level information is extracted and mapped to the appropriate target term (or composition of target terms, *cf.* the guidelines in Section 3), it is interpreted by the debugger. To step through the source code instead of the target code, the `controller` part of the debugger checks for the presence of a meta-information call at the level of the translated program (Hint 1), and displays a trace step accordingly. In particular, it is responsible for locating the name and execution counter of the target goal in the nodes corresponding to this goal, and for replacing it with the related symbolic information, *e.g.,* the name and the counter of the source-level goal associated with this target goal (Hint 2). Note that if a source-level goal maps to a composition of goals, the controller will behave as if only one step occurs, hiding the underlying target goals in the trace display either until another annotated goal is encountered, or until the last target-level goal has been (silently) executed. The information necessary to this source-level step is stored, in order to refer to it in a later step (*e.g.,* exit or failure

```
2 2 Call: ex0:f(3,_6371) ?
3 3 Call: f(3) := 3 < 42  ? k(l(m(3))) * 3 | 1000 ?
4 4 Call: f(3) := 3 < 42 ? k(l(m(3))) * 3 | 1000 ?
5 5 Call: m(3) := 3 ?
6 4 Call: f(3) := 3 < 42 ? k(l(m(3))) * 3 | 1000 ?
7 5 Call: l(3) := 3 - 2
8 4 Call: f(3) := 3 < 42 ? k(l(m(3))) * 3 | 1000 ?
9 5 Call: k(1) := 1 + 1
10 3 Call: f(3):= 3 < 42 ? k(l(m(3))) * 3  1000 ?
2 2 Exit: ex0:f(3,12) ?
```

Fig. 7. An excerpt of the debugger trace, customized with source information.

step implying backtracking) (Hint 3). When a goal invoked in the debugger is neither annotated nor part of an annotation, the controller executes its original procedure to display the standard, expanded debug information.

4.2 Source-level tracing: the functional example revisited

With this instrumentation, Example 1 is now debugged in source code terms, as illustrated in Figure 7. Note that the debugger now displays the complete declaration (see second line) defining `f`, instead of a single part of a clause (see the second line in Example 1). When a function evaluation returns a value (which is the case of all the functions `f/1`, `k/1`, `l/1`, `m/1`), intermediate unifications are performed by the generic debugger. When the debugger is instrumented with a controller (*i.e.,* the handler of annotations), these unification steps are ignored (skipped over), since they have no representation in the original source code.

5 Related Work

There exist frameworks and generative approaches that facilitate the development of DSL tools for programming, including debuggers [6,20]. For example, the Eclipse Integrated Development Environment [6], provides an API and an underlying framework that can greatly help in the development of a debugger [5]. Emacs is another example of such environments, with facilities in the same line as Eclipse. However, these tools are large and have a significant learning curve, and, more importantly, their facilities are centered more around the graphical navigation of the source code and interfacing with a command-line debugger, while the focus of our work is on bridging syntactic or semantic aspects between two sides of a translation, within such a command-line debugger. In that sense our work is complementary to (and in practice combines well with) the facilities in Eclipse, Emacs, and related environments. Generative approaches have been suggested (*e.g.,* based on aspect weaving into the language grammar [22]) in order to reduce developer burden when using intricate APIs.

However, none of these approaches provide a methodology for developing reliable and maintainable debuggers. As a result, the development of debuggers has remained difficult, inciting DSL tool developers to implement ad-hoc solutions, through extension-specific modifications and adaptations of the debugger code. For example, SWI-Prolog and Logtalk include a debugger for Prolog with built-in support for language extensions like DCGs programs [21], which is purely based on storing line numbers within the code. As mentioned in the introduction, this approach, although useful in practice, is limited to a reduced kind of extensions.

Our objective has been to develop a more general approach, which we have illustrated by applying the same methodology to several extensions including functional notation, DCGs, and CLP{Q,R}.

Lindeman *et al.* [11] have proposed recently a declarative approach to defining debuggers. To this end, they use SDF [19], a rewriting system, to instrument the abstract syntax tree with debugging annotations. However, it does not seem obvious that their approach could be applied to other observer tools. Indeed, instrumentation is achieved by providing debugger-specific information, in the form of events. In contrast, our instrumentation process makes it possible to easily add and handle different kinds of meta-information.

Unexpansion and decompilation only differ in the hypothesis used in decompilation: that the original source code may not be available. It is interesting however to compare to existing related decompilation approaches. Bowen [1] proposes a compilation process from Prolog to object code which makes it possible to define decompilation as an inverse call to compilation, provided some reordering of calls is performed. Gomez *et al.* [7] also propose a decompilation process for Java based on partial evaluation. However, these approaches have not been designed to be applicable to a large class of different language extensions. More generally, while it is in theory possible (although predictably hard with current technology) to implement fully reversible transformations, this approach runs into the problem that such inversions are non-deterministic in general, in the sense that a given target code can be generated from multiple source texts. Presenting the programmer with a different code that what is in the source program could be even more confusing that debugging the target code directly.

More similar to our solution is the approach of Tratt [18], which also targets language extensions, and where source information is injected into the abstract syntax tree of the source program. This information is exploited to report errors in terms of the language extension. However, they only discuss how to inject such information in the syntax tree, and do not explain how to use this information when building or adapting tools.

The macro-expansion passing style [4] approach makes it possible to easily implement observers. Our approach differs from this one by relying on the existing generic debugger (Ciao's in our examples): it focuses on what changes are required in the debugger and the extension framework so that symbolic information for unexpansion is handled in a way that is independent from the concrete language extension.

6 Conclusion and future work

We have presented a generic approach that enables a debugger for a target language to display trace information in terms of the language extension in which a source program is written, using the Ciao debugger as an example. The proposed approach is based on an extension of the usual mechanisms for term expansion, and in particular of their modular implementation in Ciao through *packages*. Nevertheless, we believe that our proposal could be ported to other Prolog systems with minor modifications. More specifically, we have defined a methodology for making relevant parts of the source text and other characteristics available at the target level by enriching the translation rules. We have shown that the compiler and the debugger require only small adaptations in order to take this mechanism into account; these adaptations are generic in the sense that while the transformation rules are specific to the extension, the compiler and the debugger do not require further modification, for what is arguably a large class of extensions. In particular, in the paper we have illustrated this approach by applying it on the functional notation. In the system, we have successfully applied it also to the DCG and CLP{Q,R} constraint packages.

In future work, we plan to extend the flexibility of the approach by enriching the annotations to serve different purposes, such as performing computations on symbolic information. Also, we feel that this initial work on augmenting the language extension mechanism already provides us with the basis for adapting the Ciao pre-processor. In doing so, errors, warnings, or other reports are reported in terms of the source, domain-specific language, for different extensions, without requiring further modification of the pre-processor itself. The same would apply of course to the auto-documenter and the profiler [12]. Finally, we believe we could leverage Kishon *et al.*'s framework [10] to check the soundness of our approach with regard to the intended semantics of a language extension. Doing so would also make it possible to show the equivalence between the behavior of an ad-hoc source level debugger and our customization of the target level debugger.

References

1. J. P. Bowen. From programs to object code and back again using logic programming: Compilation and decompilation. *Journal Of Software Maintenance Research And Practice*, 5(4):205–234, 1993.
2. 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.
3. 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.
4. R. K. Dybvig, D. P. Friedman, and C. T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, 1988.
5. Eclipse. How to write an Eclipse debugger. `http://www.eclipse.org/articles/Article-Debugger/how-to.html`.

6. ECRC. *Eclipse User's Guide*. European Computer Research Center, 1993.
7. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51(10):1409–1427, October 2009.
8. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012.
9. P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.
10. A. Kishon, P. Hudak, and C. Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, PLDI, pages 338–352, Toronto, Canada, 1991. ACM Press.
11. R. T. Lindeman, L. C. Kats, and E. Visser. Declaratively defining domain-specific language debuggers. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 127–136, New York, NY, USA, 2011. ACM.
12. E. Mera, T. Trigo, P. López-García, and M. V. Hermenegildo. Profiling for run-time checking of computational properties and performance debugging in logic programs. In *Practical Aspects of Declarative Languages*, PADL '11, pages 38–53, 2011.
13. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.
14. J. F. Morales, M. V. Hermenegildo, and R. Haemmerlé. Modular Extensions for Modular (Logic) Languages. In *Proceeding of the 21th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'11)*, volume 7225 of *LNCS*, pages 139–154, Odense, Denmark, July 2012. Springer.
15. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction*, pages 138–152, 2003.
16. F. Pereira and D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.
17. A. P. Tolmach and A. W. Appel. A debugger for standard ml. *J. Funct. Program.*, 5(2):155–200, 1995.
18. L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):31:1–31:40, Oct. 2008.
19. M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. In *Compiler Construction*, pages 365–370, 2001.
20. M. G. J. van den Brand, B. Cornelissen, P. A. Olivier, and J. J. Vinju. Tide: A generic debugging framework — tool demonstration —. *Electron. Notes Theor. Comput. Sci.*, 141(4):161–165, Dec. 2005.
21. J. Wielemaker. SWI-prolog — source-level debugger. `http://www.swi-prolog.org/gtrace.html`.
22. H. Wu, J. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 1370–1374, New York, NY, USA, 2005. ACM.