Architecturing Software Using A Methodology for Language Development

Charles Consel and Renaud Marlet

IRISA / INRIA - University of Rennes 1
Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
http://www.irisa.fr/compose
{consel,marlet}@irisa.fr

1 Introduction

Domain-specific languages (DSLs) can be viewed from both a programming language and a software architecture perspective. The goal of this paper is to relate the two viewpoints. In particular, we demonstrate that DSLs can be constructed using an existing formal methodology for developing general purpose languages (GPLs) while expressing software architecture concerns.

1.1 A Programming Language Perspective

A DSL can be viewed as a programming (or specification) language dedicated to a particular domain or problem. It provides appropriate built-in abstractions and notations; it is usually small, more declarative than imperative, and less expressive than a GPL.

Consider for example the Unix command `make`. This tool is a utility to maintain programs: it determines automatically which pieces of a large program need to be recompiled, and issues the commands to recompile them. The language of makefiles is small (at least in the early versions of `make`) and mainly declarative, although it also contains some imperative constructs. Its expressive power is limited to updating task dependencies; actual recompilation actions are delegated to a shell. It hides implementation details like file last-modification time and provides domain abstractions such as file suffixes and implicit compilation rules. As a result, the user may concisely express precise update dependencies.

This example illustrates several important DSL features, which make DSLs more attractive than GPLs for a variety of applications.

Easier programming. Because of appropriate abstractions, notations and declarative formulations, a DSL program is more concise and readable than its GPL counterpart. Hence, development time is shortened and maintenance is improved. As programming focuses on *what* to compute as opposed to *how* to compute, the user does not have to be a skilled programmer. For example, in the case of recompilation, writing a program to explicitly test all file modification times in order to incrementally rebuild a system would clearly be lengthy, tedious and error-prone.

Systematic re-use. Most GPL programming environments include the ability to group common operations into libraries. Though some are standard libraries, re-use is left to the programmer. On the other hand, a DSL offers guidelines and built-in functionalities which enforce re-use. Additionally, a DSL captures domain expertise, either implicitly by hiding common program patterns in the DSL implementation, or explicitly by exposing appropriate parameterization to the DSL programmer. Thus, any user necessarily re-uses library components and domain expertise.

Easier verification. Another important advantage of DSLs is that they enable more properties about programs to be checked. In contrast to a GPL, the semantics of a DSL can be restricted to make decidable some properties that are critical to a domain. For example, `make` reports any cycle in dependencies and thus totally prevents non-termination (assuming the individual actions do not loop).

Although all DSL features listed above address important software engineering concerns, they do not say much about the way applications based on DSLs should be structured. In fact, DSLs strongly suggest particular software architectures.

1.2 A Software Architecture Perspective

Software architectures express how systems should be built from various components and how those components should interact. From a software architecture perspective, a DSL can be seen as a parameterization mechanism as well as an interface model.

Parameterization mechanism. A program or a library can be more or less generic depending on the scope of the problems it addresses [3]. For example, a scientific library can be highly generic considering the vast variety of problems it can be used for. Pushing the idea of genericity further leads to complex parameters that can be seen as DSLs. For example, the format string argument of function `printf` can be seen as both a complex parameter and a very simple DSL. Considering a DSL program as a complex argument to a highly parameterized component may sound contrived but it actually is the final step of a chain of increasingly expressive power in parameterization. This situation is illustrated by Unix commands `grep`, `sort`, `find`, `sed`, `make`, `awk`, etc., and the progression from simple command-line parameters to program files. At the end of the spectrum, the data parameter ends up being a program to be processed, yielding increased parameterization power.

Interface to a library. As a library becomes larger or more generic, its usability decreases due to the multiplication of entry points, parameters and options offered. As a result, the library might be ignored by programmers because it is too complex to use. In this situation, a DSL can offer a domain-specific interface to the library so that the programmer does not have to directly manipulate numerous highly-parameterized building blocks; the complexity is hidden. Another common situation is when some patterns of library calls occur frequently. In this case, a DSL interface can provide direct access to those commonly used combinations. For example, Unix shells are interfaces to standard Unix libraries. This idea is shared by scripting languages, that glue together a set of powerful components written in traditional programming languages. For example, Tcl/Tk provides a Tcl interface on top of the Tk graphic toolkit.

Recognizing a DSL as both a parameterization mechanism and an interface has an impact on structuring and reasoning about the software. In fact, the range of software adaptability is defined by the DSL. Such software is thus naturally separable into two parts: the decoding of the parameterization expressed by DSL programs and a library of components.

One may wonder when complex parameters and library interfaces are used. In the first case, complex parameters are introduced when, instead of offering separate but related tools, a single, versatile program is provided. In the second case, libraries are in essence created to enable re-use of data types and basic operations among related programs. Thus, the common motivation of those architectures is to build a set of related programs. This observation leads us to another, more conceptual aspect of DSLs: a program family.

Program family. A DSL program designates a member of a program family. A *program family* is a set of programs that share enough characteristics that it is worthwhile to study them as a whole [26]. A program family can also be seen as providing a solution to a *problem family*, *i.e.*, a set of related problems. Drivers, for a given type of device, form a natural example of a program family: in addition to having the same API (for a given operating system), they all share similar operations, although they vary according to the hardware.

1.3 When to develop a DSL?

Conversely, we believe that whenever a problem family must be solved, *i.e.*, whenever a program family must be developed, basing the software architecture on a DSL makes configuration (*i.e.*, DSL programming) simpler. More generally, the following issues should be raised even when developing new software: does the program to be developed address an isolated problem? Could it be a member of a future program family?

The fact is that existing DSLs do implement program families. Examples are numerous; DSLs have been used in various domains such as graphics [12, 19], financial products [1], telephone switching systems [13, 21], protocols [4, 34], operating systems [29], device drivers [36], routers in networks [34] and robot languages [2]. This profusion also shows the recent attention that DSLs have received from both the research and industrial communities.

1.4 How to develop a DSL?

These applications have clearly illustrated the advantages of DSLs over GPLs, recording benefits such as productivity, reliability and flexibility [20]. However, they also raise a key issue which, if not addressed, could obstruct the use of DSLs [5]: how does one design and implement a DSL? Resolving this issue is critical to make the approach profitable since there is no point in reducing the complexity of program development by shifting all the complexity into the construction and maintenance of a DSL programming environment.

Another related question is: who will develop DSLs? Even in the programming language community, only a few people have actually designed a language. *A fortiori*, we cannot expect software engineers to have the full expertise to build up new languages. Thus, it is crucial that a methodology and tools are provided to make the DSL approach widely accessible.

1.5 Our Methodology for Developing DSLs

We propose a methodology for designing and implementing DSLs. This methodology is based on an existing formal framework for defining GPLs and integrates software architecture concerns.

This formal framework is based on *denotational semantics*, which has been extensively used to formally define GPLs [31]. It identifies key concepts in language design and semantics. Furthermore, techniques have been developed to derive implementations from definitions in denotational semantics [14]. These techniques typically produce compilers that are less efficient than ad hoc GPL compilers. However, in the context of DSLs, efficiency often relies on the underlying building blocks. As will be shown in this paper, structuring a DSL definition allows these building blocks to be isolated and implemented efficiently.

Our methodology is based on a framework outlined in an earlier paper by Thibault and Consel [33]. It can be summarized as follows. For the sake of clarity, the phases of our methodology are presented sequentially. In practice, the whole process needs to be *iterated*. Notice that the working example used to illustrate each phase throughout the paper is the final result of this iteration.

Language analysis. Assuming a problem family has been identified, the first step is to analyze the commonalities and the variations in the corresponding program family. This analysis is fueled by domain knowledge. The result of this analysis includes a description of objects and operations that are needed to express solutions to the family of problems, as well as language requirements (*e.g.*, analyzability) and elements of design (*e.g.*, notations).

Interface definitions. The next phase is to refine the design elements of the DSL. To do so, the syntax of the DSL is defined and its informal semantics is developed. The informal semantics relates the syntactic constructs to the objects and operations (*i.e.*, the building blocks) identified previously. Additionally, the domain of objects and the type of operations are formalized, thus forming the signature of semantic algebras.

Staged semantics. The semantics of a GPL is typically split between the compile-time and the run-time actions. These two parts are also referred to as the *static* and the *dynamic semantics* of a language. We propose to perform the same separation in the semantics of a DSL. With respect to software architecture concerns, this separation makes stages of configuration explicit.

Formal definition. Once the static and dynamic components of the language have been determined, the DSL is formally defined. Valuation functions define the semantics of the syntactic constructs. They specify how the operations of the semantic algebras (*i.e.,* the building blocks) are combined.

Abstract machine. Then, the dynamic semantic algebras are grouped to form a dedicated abstract machine which models the dynamic semantics of the DSL. From the denotational semantics, the DSL is given an interpretation in terms of this abstract machine. The state of the semantics is globalized and mapped into abstract machine entities (*e.g.*, registers) dedicated to the program family.

Implementation. The abstract machine is then given an implementation (typically, a library), or possibly many, to account for different operational contexts. The valuation function can be implemented as an interpreter based on an abstract machine implementation, or as a compiler to abstract machine instructions.

Partial evaluation. While interpreting is more flexible, compiling is more efficient. To get the best of both worlds, we use a program transformation technique, namely, partial evaluation, to automatically transform a DSL program into a compiled program, given only an interpreter.

Each of the above methodology steps is further detailed in a separate section of this paper.

1.6 A Working Example

To illustrate our approach, an example of DSL is used throughout the paper. We introduce a simple electronic mail processing application as a working example. Conceptually this application enables users to specify automatic treatments of incoming messages depending on their nature and contents: dispatching messages to people or folders, filtering spam, offering a shell escape (*e.g.*, to feed an electronic agenda), replying to messages when absent, etc.

This example is inspired by a Unix program called `slocal` which offers users a way of processing inbound mail. With `slocal`, user-defined treatments are expressed in the form of rules. Each rule consists of a string to be searched in a message field (*e.g.*, `Subject`, `From`) and an action to be performed if the string is found. Each rule stands on a single line and the whole specification is a flat series of rule lines, as opposed to a structured program.

This simple application illustrates the situation where a family of problems has to be handled: addressing different needs for the treatment of messages. One could imagine a combination of various GPL programs being written to address each kind of treatment. This would form a family of programs which would most likely rely on a dedicated library. This library would consist of basic operations such as parsing a message, accessing and modifying message header fields, archiving and sending messages, etc.

We present a DSL solution to this problem family. More precisely, we show how to design and implement MAILSH, a simple DSL aimed at specifying the automatic treatment of incoming e-mails. Some details are left out of the following discussion. Our goal is to illustrate our methodology, not to propose an alternative to `slocal`.

2 Language Analysis

In the first phase of our approach, we analyze the problem family. During this analysis, the commonalities (shared features and assumptions that hold for all family members) and variabilities (variations in behavior and assumptions that differ among family members) must be identified. The analysis takes into account domain knowledge such as technical literature, existing programs, and current and future requirements. It can be conducted using methodologies used for *commonality analysis*, such as FAST [39, 11], and *domain analysis* [24, 25, 28]. The main results of this analysis phase are: language requirements, a description of the common objects and operations, and design elements of the DSL. We examine each of these items in turn and illustrate them with our working example.

2.1 Language Requirements

Analyzing the family of problems leads to requirements for the language. Those requirements mainly consist of the functionalities that must be expressible in the DSL. Requirements also include language constraints (*e.g.*, domain issues such as safety and security) and implementation constraints (*e.g.*, resource bounds).

This phase does not differ much from a problem analysis that occurs when initiating any software development. The difference is that requirements are expressed in terms of language issues rather than general features of the application.

Working example. Concerning our message processing application, it should be possible, at the language level, to copy, move, delete, forward, pipe to a shell command, and reply to a message. Those actions should be triggered according to conditions depending on the inbound message. Those conditions should be string patterns matched against fields of the message.

Moreover, we have determined four language constraints. First, the user-defined treatments determined by a MAILSH program should not loop. Second, treatments should be guaranteed not to lose inbound messages. Third, inbound messages should not be duplicated in the same folder when archived. Fourth, automatically forwarding messages should not cause endless loops.

2.2 Objects and Operations

Identifying the common objects and operations essentially corresponds to defining the basic building blocks needed to express solutions for the family of problems. From a software architecture point of view, this process can be viewed as designing a library since it captures the common program patterns in the family and abstracts over the differences. The building blocks are grouped with respect to the objects they manipulate.

Working example. The program family analysis of our e-mail processing application results in the following fundamental objects and operations.

Messages. An electronic message consists of header fields and a body. We need operations to manipulate these message fields and to create new messages.

Folders. Folders contain a list of messages. Assuming we limit ourselves to dispatching messages, the only operation needed is to add a message to a folder.

Hierarchies of folders. A user typically has many folders to which (s)he directs messages, *e.g.*, according to topic or source. To cope with an increasing number of folders, e-mail systems offer the ability to create a folder hierarchy. To treat this feature in our system we need to associate an actual filename to a folder path in the folder hierarchy.

Files of Folders. Because of the layer introduced by the hierarchy of folders, the actual folders need to be captured by a separate object. Operations to read and write a folder from/to the file system need to be introduced.

Streams. Messages need to be sent, received or piped into a shell command. To model this, we need streams of inbound and outbound messages, as well as a command stream.

Miscellaneous. There are other, less fundamental objects and operations that we do not further detail here. This includes the ability to know the user's name (to send messages) and the current date (to timestamp the messages). There are also operations on booleans and strings, in particular a pattern matcher used in the message filtering condition.

2.3 Elements of Design

The last part of the language analysis phase consists of determining elements of the language design. These elements include the language paradigm (*e.g.*, declarative or imperative) as well as the language level: from low-level for expressivity, to high-level for usability. In addition, a terminology and notations are developed both from the domain and the set of problems to be addressed. These notations must correspond to the way domain experts express a solution, *i.e.*, a member of the problem family.

Working example. To apply this phase to our example, we have to introduce assumptions about the users of this message processing system. We assume such users to be typical Unix shell programmers. As a result, we decide the DSL should be imperative like shell languages. Moreover, selection criteria should include regular expressions to achieve pattern matching in messages, as provided in the shell languages.

3 Interface Definitions

Given the information collected previously, we are now ready to develop a preliminary specification of the DSL. This preliminary specification consists of defining interfaces: the signature of semantic algebras and the DSL syntax. The semantics is kept informal; it will be made explicit in a later phase (see Section 5). Still, it allows taking into account some language requirements and to prepare the structuring of the actual language definition.

3.1 Semantic Algebras

The common objects and operations collected in the previous phase are now grouped with respect to the objects they manipulate to produce abstract data types. In the denotational framework, this form of abstract data types can be formalized as semantic algebras. A semantic algebra formally defines a domain (*i.e.*, a structured value space) and the operators on that domain [31]. At this stage, we only provide signatures; we postpone details until a complete view of basic building blocks is determined.

Working example. Let us illustrate the notion of semantic algebra with our message processing application. To do so, we present in Figure 1 the signature of semantic algebras which follow the common objects and operations determined earlier in Section 2.1.

Messages. Function *msg-to-string* converts a message into a string. This function is used when piping a message into a shell command and when forwarding a message. In the latter case, the body of the new message (a string) contains the forwarded message. *FieldName* is defined as *String*.

Folders. We consider a folder as an ordered list of messages; function *add-msg* adds a message at the end of this list. There are other obvious common operations on folders; we do not mention them here as they are not needed for our example.

Hierarchies of folders. Function *get-filename* maps a folder path into a filename. Note that the folder hierarchy may define aliases: two paths may be mapped into the same filename.

Files of Folders. There are several common implementations of a folder, depending on the user's mailing system. We let actual implementations of abstract operators *read-folder* and *write-folder* deal with that.

Messages
Domain: *Message*
Operations:
$\mathit{new\text{-}msg} : \mathit{Message}$
$\mathit{get\text{-}field} : \mathit{FieldName} \to \mathit{Message} \to \mathit{String}$
$\mathit{set\text{-}field} : \mathit{FieldName} \to \mathit{String} \to \mathit{Message} \to \mathit{Message}$
$\mathit{get\text{-}body} : \mathit{Message} \to \mathit{String}$
$\mathit{set\text{-}body} : \mathit{String} \to \mathit{Message} \to \mathit{Message}$
$\mathit{msg\text{-}to\text{-}string} : \mathit{Message} \to \mathit{String}$

Folders
Domain: *Folder*
Operations:
$\mathit{add\text{-}msg} : \mathit{Message} \to \mathit{Folder} \to \mathit{Folder}$

Hierarchy of Folders
Domain: *FolderHierarchy*
Operations:
$\mathit{get\text{-}filename} : \mathit{FolderPath} \to \mathit{FolderHierarchy} \to \mathit{FileName}$

Files of Folders
Domain: *FolderFiles*
Operations:
$\mathit{read\text{-}folder} : \mathit{FileName} \to \mathit{FolderFiles} \to \mathit{Folder}$
$\mathit{write\text{-}folder} : \mathit{FileName} \to \mathit{Folder} \to \mathit{FolderFiles} \to \mathit{FolderFiles}$

Streams
Domains: *InStream*, *OutStream*, *CmdStream*
Operations:
$\mathit{next\text{-}msg} : \mathit{InStream} \to (\mathit{Message} \times \mathit{InStream})$
$\mathit{send\text{-}msg} : \mathit{Message} \to \mathit{OutStream} \to \mathit{OutStream}$
$\mathit{pipe\text{-}msg} : \mathit{Message} \to \mathit{CmdString} \to \mathit{CmdStream} \to \mathit{CmdStream}$

Fig. 1. Signature of the main semantic algebras for MAILSH

$$\begin{array}{lcl}
B & \in & \mathit{BoolExpr} \\
C & \in & \mathit{Command} \\
F & \in & \mathit{FolderPath} \\
S & \in & \mathit{String} \\
\\
B & ::= & \mathtt{match}\ S_{\mathsf{field}}\ S_{\mathsf{pat}} \\
 & \mid & \mathtt{not}\ B \mid B_1\ \mathtt{and}\ B_2 \mid B_1\ \mathtt{or}\ B_2
\end{array}
\qquad
\begin{array}{lcl}
C & ::= & C_1\ \mathtt{;}\ C_2 \\
 & \mid & \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \\
 & \mid & \mathtt{skip} \\
 & \mid & \mathtt{delete} \\
 & \mid & \mathtt{copy}\ F \\
 & \mid & \mathtt{forward}\ S_{\mathsf{to}} \\
 & \mid & \mathtt{reply}\ S_{\mathsf{body}} \\
 & \mid & \mathtt{pipe}\ S_{\mathsf{cmd}}
\end{array}$$

Fig. 2. Abstract syntax of MAILSH

Streams. Function *next-msg* reads the next incoming message. Note that an implementation of it must not return until a new message has arrived, thus suspending the application. Function *send-msg* ships a message to the system stream. Function *pipe-msg* passes a string to the standard input of a command. We also define *CmdString* as *String*.

Miscellaneous. We do not detail here other miscellaneous semantic algebras. We will later only explicitly use $match : String \to StringPattern \to Bool$ as the pattern matching operator.

3.2 The DSL Syntax

The syntax of the DSL is defined based on information collected earlier, namely, the language requirements (functionalities as well as constraints) and the design elements (language paradigm, language level, terminology and notations). It may also explicitly refer to some of the objects and operations identified as fundamental; the others remain hidden in the underlying semantics. Intuitively, a syntactic construct in the DSL corresponds to a pattern of operations.

One of the key issues in designing the abstract syntax (*i.e.*, the interface of the language) is to restrict the programmability so that required properties are provable. At the same time, raising the level of the DSL may hinder future needs for expressiveness. For example, a common practice to ensure the termination of DSL programs is to provide the programmer only with restricted loop constructs, if any, so that the property can be syntactically checked. Issues regarding the design of a concrete syntax are beyond this work.

As the DSL syntax is developed, its semantics is informally defined. This preliminary definition allows the semantic algebras to be further refined.

Working example. The requirements were that MAILSH should express conditional treatment of incoming messages, be imperative and close to Unix shells. Figure 2 presents the BNF definition of an abstract syntax which fulfills those requirements. Folders are an example of a domain-specific object explicitely referred to by the syntax via folder paths. In contrast, folder filenames remain hidden. For the sake of simplicity, we have intentionally reduced this DSL to a kernel language, rich enough to allow us to illustrate the various aspects of our approach. Obviously, to make it usable, more constructs and actions on messages should be added. For example, the following abbreviations could be provided:

– `move` F $\equiv$ `copy` F `; delete`
– `if` B `then` C $\equiv$ `if` B `then` C `else skip`

A concrete syntax close to Unix shells can easily be developed.

We shall not comment here on the semantics of the various constructs of the language since most of them are self-explanatory. An example program is given in Figure 3. (Indentation emphasizes the nesting of constructs.) Note that the `reply` construct can be used to setup a `vacation`-like tool, *i.e.*, a message-sensitive answering machine.

```
if match "Subject" "DSL" then
  forward "jake";
    copy Research.Lang.DSL; delete
else if match "From" "hotmail.com" then
       reply "Leave me alone!"; delete
     else if match "Subject" "seminar" then
            pipe "agenda --stdin"; delete
          else
            skip
```

Fig. 3. Abstract syntax of a MAILSH program

It must be noted that the language only specifies the treatment of a single message; there is an implicit loop over the inbound messages. This kind of treatment encapsulation is typical to DSLs. Common examples are text processing DSLs, like `sed`, that assume an implicit loop over each line of the text input.

Before providing a formal definition for the language and tackling its implementation, a staging phase is required to separate the language semantic entities.

4 Staged Semantics

From a programming language viewpoint, the semantics of a GPL is traditionally split into two parts: the static and the dynamic semantics. In practice, the static semantics of a language corresponds to the actions performed by a compiler; these actions are thus the ones which depend on the program being compiled. The dynamic semantics represents the computations which may depend on the input data of the program. Necessarily, these computations must be postponed until run time.

Because a compiler processes the static semantics of a language with respect to a given program, it is in effect a syntax-to-dynamic-semantics mapping [10]. Concretely, the dynamic semantics is a compiled program; it consists of a combination of instructions for a machine (either abstract or concrete).

From a software architecture viewpoint, the static semantics corresponds to computations that determine the member of a program family. The dynamic semantics corresponds to computations that produce the answer to the corresponding problem, *i.e.*, program execution.

From an implementation viewpoint, processing the static semantics of an application can be seen as configuring generic software with respect to a given context. More concretely, configuring amounts to processing the static (*i.e.*, available) information in order to select the appropriate components, and combining them to produce a customized software. Then, processing the dynamic semantics consists of executing the customized software.

Determining staging addresses an important concern in software architecture: reasoning about the genericity of software to predict and control its customization. This is a key step towards reconciling flexibility, as promoted by many approaches to software architecture, and performance. Indeed, inefficiency is a well-known limitation of many of these approaches [23].

We propose to address the staging of a DSL semantics, or equivalently the configuration of its software architecture counterparts, using a language approach. At this point of our methodology, the staging process is limited to semantic algebras, instead of being applied to the complete DSL definition. Later, when providing an actual definition (the valuation functions), more staging issues will also have to be considered. For the moment, from initial staging constraints coming from the problem family, we introduce staging in the semantic algebras and in the treatment of language constraints.

4.1 Initial Staging Constraints

To achieve the staging of a DSL semantics, the initial step is to determine the *semantic arguments* of the valuation functions which can be assumed to be static (*i.e.* known) given the problem family. These static arguments can be seen as configuration arguments. Just like GPLs, the static semantics of DSLs assumes that the program is available (*i.e.*, static).

Working example. Considering our message processing application, we assume that the static arguments are the DSL program representing the user-defined treatments, the folder hierarchy of the user, and the user's name.

4.2 Staging the Semantic Algebras

Given this initial staging, the semantic algebras need to be analyzed to determine which ones correspond to configuration (*i.e.*, static) computations and which ones define *actual* (*i.e.*, dynamic) computations. For example, in the context of a GPL, a semantic algebra which maintains type information on program variables is typically a static algebra when the language is strongly typed.

For a given DSL, the staging process should answer the following question on each semantic algebra: should this value domain, and its corresponding operations, be static or dynamic?

Working example. Given that the folder hierarchy is a static initial argument to MAILSH, it should remain unchanged throughout the semantics since our DSL does not provide a way to modify this hierarchy. Therefore, the semantic algebra for the hierarchy of folders should be static as well; operations on such values should be processed completely statically.

The other semantic algebras of our DSL are intrinsically dynamic since they rely on values assumed to be known only at run time, *i.e.*, inbound messages.

4.3 Staging the Language Constraints

Staging not only involves the language semantics but also the language constraints, which in turn has an impact on the semantics. Some constraints may be guaranteed statically, before the DSL program is run. Others may rely on run-time information and have to be checked when the DSL program is executed.

Working example. The first language constraint (see Section 2.1) is that the treatment of a message should not loop. This constraint is syntactically enforced given that there is no iteration construct.

The second constraint states that user-defined treatments should be guaranteed not to lose inbound messages. An inbound message is lost when it is neither copied, forwarded, piped, replied-to nor explicitly deleted, *i.e.*, when it is skipped. For example, the program in Figure 3 can lose a message if none of the conditions applies. This constraint can be statically checked by analyzing the possible execution paths of a program with respect to the pattern matching conditions. (Proof omitted.)

The third language constraint can also be statically checked. (Proof omitted.) It states that inbound messages should not be duplicated in the same folder when archived.

The fourth language constraint says that automatically forwarding messages should not be able to cause endless loops. Because of unknown aliases and mailing lists, it is not possible to make sure that, if a message is forwarded, it will not eventually be forwarded back to the sender. This condition can only be checked dynamically by introducing a specific mechanism.

An additional constraint has not been expressed yet because it depends more on the structure of the language than on the domain: it should not be possible to operate on a message after it has been deleted. This amounts to checking paths where there exist message treatments after a `delete` invocation. This property can be statically checked. (Proof omitted.) We make the decision to reject any program not satisfying this property. As we will see, not only does it prevent us from specifying error handling in the dynamic semantics, but it also simplifies the implementation of `delete`, turning it into a mere `skip`.

5 Formal Definition

We now have all the necessary elements to formally define the semantics of a DSL. Fundamentally, the denotational definition represents a guide for the language implementer and a key source of documentation. By postponing implementation issues to a later phase, the DSL developer can better stage decisions. For example, the data layout of objects can be postponed until hardware features are known.

As is customary, the denotational semantics is composed of three parts: the abstract syntax (see Section 3.2), the semantic algebras (see Section 3.1), and the valuation functions. In contrast to the informal semantics given previously, the semantic algebras are now completely specified, including the definition of their operations.

5.1 Semantic Arguments

Valuation functions are inductively defined on the abstract syntax. Besides the program text, a valuation function includes other semantic arguments which define the semantic context. The semantic arguments are drawn from the semantic algebras introduced earlier.

Working example. The semantic arguments in the case of MAILSH are the folder hierarchy, the message being treated, the folder files, the streams and other miscellaneous entities like the current date and the user's name.

5.2 Staging the Semantic Arguments

Beyond the semantic algebras, the valuation functions must further separate the DSL semantics into its static and dynamic parts. To do so, we keep separate the static and dynamic semantic arguments of the valuation functions. This separation is guided by the binding time (static / dynamic) of the semantic algebras determined previously.

Working example. As for MAILSH, the static semantic arguments are the folder hierarchy and the user's name; these are grouped into a product domain named *StaticState*. The dynamic arguments are the message being treated, the folder files, the streams and the current date.

We use the following notations: the tuple projection on the domain X (*e.g.*, *Message*) of $\sigma \in \mathit{DynamicState}$ is denoted σ_{x} (*e.g.*, $\sigma_{\mathsf{message}}$). Updating the X element of the tuple σ with a value y is denoted $[\mathsf{x} \mapsto y]\,\sigma$.

5.3 Control Staging and Dynamic Combinators

The computations described by the valuation functions also need to be staged. The basic operations used by a valuation function have a binding time that has been determined in the previous phase; only the control operations remain to be staged. To do so, the separation between static and dynamic control operations must be made explicit. We thus introduce combinators for the dynamic control operations; these combinators are later turned into control instructions in the abstract machine. Static control operations need not be associated with an explicit combinator.

Working example. The conditional statement `if` B `then` C_1 `else` C_2 is dynamic because it depends on the message to be treated via the `match` construct. We thus introduce a choice function as an explicit *cond* combinator.

5.4 Valuation Functions

The valuation functions may finally be defined. Complete definition of the semantic algebras should be provided at this stage as well.

$$\mathcal{C} : \mathit{Command} \to \mathit{StaticState} \to \mathit{DynamicState} \to \mathit{DynamicState} \quad \text{where}$$
$$\mathit{StaticState} = \mathit{FolderHierarchy} \times \mathit{UserName}$$
$$\mathit{DynamicState} = \mathit{FolderFiles} \times \mathit{OutStream} \times \mathit{CmdStream} \times \mathit{Date} \times \mathit{Message}$$

$$\mathcal{C}\ [\![C_1\ \texttt{;}\ C_2]\!]\ \rho = (\mathcal{C}\ [\![C_2]\!]\ \rho) \circ (\mathcal{C}\ [\![C_1]\!]\ \rho)$$
$$\mathcal{C}\ [\![\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\ \rho = \mathit{cond}\ (\mathcal{B}\ [\![B]\!])\ (\mathcal{C}\ [\![C_1]\!]\ \rho)\ (\mathcal{C}\ [\![C_2]\!]\ \rho)$$
$$\mathcal{C}\ [\![\texttt{skip}]\!]\ \rho\ \sigma = \sigma$$
$$\begin{array}{l}
\mathcal{C}\ [\![\texttt{copy}\ F]\!]\ \rho\ \sigma = \\
\quad \text{let}\ \nu = \mathit{get\text{-}filename}\ (\mathcal{F}\ [\![F]\!])\ \rho_{\mathsf{folder\text{-}hierarchy}} \\
\qquad \varphi = \mathit{add\text{-}msg}\ (\mathit{set\text{-}field}\ \text{“}\texttt{Delivery-Date}\text{”}\ \sigma_{\mathsf{date}}\ \sigma_{\mathsf{message}}) \\
\qquad\qquad\qquad (\mathit{read\text{-}folder}\ \nu\ \sigma_{\mathsf{folder\text{-}files}}) \\
\quad \text{in}\ [\mathsf{folder\text{-}files} \mapsto \mathit{write\text{-}folder}\ \nu\ \varphi\ \sigma_{\mathsf{folder\text{-}files}}]\ \sigma
\end{array}$$
$$\begin{array}{l}
\mathcal{C}\ [\![\texttt{forward}\ S]\!]\ \rho\ \sigma = \\
\quad [\mathsf{out\text{-}stream} \mapsto \mathit{send\text{-}msg} \\
\qquad (\mathit{set\text{-}field}\ \text{“}\texttt{Resent-by}\text{”}\ (\mathit{concat}\ \rho_{\mathsf{user\text{-}name}}\ (\mathit{get\text{-}field}\ \text{“}\texttt{Resent-by}\text{”}\ \sigma_{\mathsf{message}})) \\
\qquad\ (\mathit{set\text{-}field}\ \text{“}\texttt{Subject}\text{”}\ (\mathit{concat}\ \text{“}\texttt{Fwd: }\text{”}\ (\mathit{get\text{-}field}\ \text{“}\texttt{Subject}\text{”}\ \sigma_{\mathsf{message}})) \\
\qquad\quad (\mathit{set\text{-}body}\ (\mathit{msg\text{-}to\text{-}string}\ \sigma_{\mathsf{message}}) \\
\qquad\qquad (\mathit{set\text{-}field}\ \text{“}\texttt{From}\text{”}\ \rho_{\mathsf{user\text{-}name}} \\
\qquad\qquad\ (\mathit{set\text{-}field}\ \text{“}\texttt{To}\text{”}\ (\mathcal{S}\ [\![S]\!]) \\
\qquad\qquad\quad (\mathit{set\text{-}field}\ \text{“}\texttt{Date}\text{”}\ \sigma_{\mathsf{date}}\ (\mathit{new\text{-}msg})))))))\ \sigma_{\mathsf{out\text{-}stream}}]\ \sigma
\end{array}$$
$$\begin{array}{l}
\mathcal{C}\ [\![\texttt{reply}\ S_1]\!]\ \rho\ \sigma = \\
\quad [\mathsf{out\text{-}stream} \mapsto \mathit{send\text{-}msg} \\
\qquad (\mathit{set\text{-}field}\ \text{“}\texttt{Subject}\text{”}\ (\mathit{concat}\ \text{“}\texttt{Re: }\text{”}\ (\mathit{get\text{-}field}\ \text{“}\texttt{Subject}\text{”}\ \sigma_{\mathsf{message}})) \\
\qquad\ (\mathit{set\text{-}body}\ (\mathcal{S}\ [\![S_1]\!]) \\
\qquad\quad (\mathit{set\text{-}field}\ \text{“}\texttt{From}\text{”}\ \rho_{\mathsf{user\text{-}name}} \\
\qquad\qquad (\mathit{set\text{-}field}\ \text{“}\texttt{To}\text{”}\ (\mathit{get\text{-}field}\ \text{“}\texttt{From}\text{”}\ \sigma_{\mathsf{message}}) \\
\qquad\qquad\ (\mathit{set\text{-}field}\ \text{“}\texttt{Date}\text{”}\ \sigma_{\mathsf{date}}\ (\mathit{new\text{-}msg}))))))\ \sigma_{\mathsf{out\text{-}stream}}]\ \sigma
\end{array}$$
$$\mathcal{C}\ [\![\texttt{pipe}\ S]\!]\ \rho\ \sigma = [\mathsf{cmd\text{-}stream} \mapsto \mathit{pipe\text{-}msg}\ \sigma_{\mathsf{message}}\ (\mathcal{S}\ [\![S]\!])\ \sigma_{\mathsf{cmd\text{-}stream}}]\ \sigma$$

$$\mathcal{B} : \mathit{BoolExpr} \to \mathit{DynamicState} \to \mathit{DynamicState}$$
$$\mathcal{B}\ [\![\texttt{match}\ S_1\ S_2]\!]\ \sigma = \mathit{match}\ (\mathit{get\text{-}field}\ (\mathcal{S}\ [\![S_1]\!])\ \sigma_{\mathsf{message}})\ (\mathcal{S}\ [\![S_2]\!])$$

Fig. 4. Valuation functions for MAILSH

Working example. Remember that the processing of messages is always active and should be modeled by an infinite loop aimed at polling the stream of inbound messages. This loop must further rely on function *next-msg*, which suspends the message processing application if no inbound message is available. When some messages are received, the dynamic semantic arguments are set up and the valuation function $\mathcal{C}$ is applied to the program, *i.e.*, a possibly structured command.

Figure 4 shows the definition of valuation function $\mathcal{C}$. The `delete` construct does not appear in the definition of $\mathcal{C}$ because it is replaced by a `skip` after analysis (see Section 4). The definition of the other valuation functions and the semantic algebras of our DSL are omitted since they are rather simple and do not raise issues with respect to our approach. Setting the "`Resent-by`" field when forwarding allows the encapsulating loop to discard incoming messages that have already been forwarded by the user, thus dynamically verifying the fourth constraint expressed in the language requirements.

Common dynamic patterns of operations in the right-hand side of the semantic equations can be encapsulated into new operators. For example, composing a message for the forward or reply operations shares dynamic operations that could have been grouped into a single higher-level operator.

6 Abstract Machine

Although the valuation functions make a clear separation between the static and dynamic semantics of the DSL, we still have to further encapsulate the dynamic semantics to define a dedicated abstract machine. This is a key step to derive a realistic implementation from the DSL definition. The abstract machine roughly corresponds to the library in a conventional software architecture. However, it is not yet the implementation (see Section 7).

Another benefit of the approach is that it provides a formal model of computation that can be reasoned about using well-established techniques for abstract machines [27]. In fact, the abstract machine offers a model of computation that underlies all programs in the family [5]. The abstract machine model also provides the right level of decomposition to increase reuse of the abstract machine [38]. In particular, since an abstract machine can express a wide range of applications within the domain, and a DSL only a restricted subset of these, several DSLs could share the same abstract machine. For example, it is useful to have multiple DSLs for different users; a DSL could thus manage a whole database while a subset of this DSL might only be able to express queries.

6.1 Single-Threadedness and Globalization

The key issue in expressing a semantics in terms of an efficient abstract machine is the globalization of the dynamic semantic arguments. To enable semantic arguments to be made implicit in the actual implementation, they cannot be manipulated in an arbitrary way by the denotational definition. Schmidt and others have developed specific criteria which allow semantic arguments to be globalized when deriving an implementation from a denotational definition [31]. If these criteria are fulfilled by the denotational definition for a given semantic argument, then the denotational definition is said to be *single-threaded* in this semantic argument. A precise definition of these criteria is beyond the scope of this paper.

If a semantics definition is single-threaded in a dynamic state argument, this argument can be globalized. For example, in case of an imperative GPL, the store is a typical semantic argument which gets globalized in an implementation of its dynamic semantics. Indeed, the store corresponds to the processor memory and thus does not need to be passed explicitly since it is globally available. In the case of a DSL, there may be various semantic arguments which need to be globalized in an actual implementation. This is one of the aspects which reflects the dedicated nature of the abstract machine of DSL.

Note that, when the dynamic state is globalized, abstract machine instructions which perform a state transition are *linearized*.

Working example. The semantics example given in Figure 4 is already single-threaded. Thus, the dynamic semantics arguments can be globalized.

6.2 Abstract Machine Entities

Our goal is to develop an abstract machine dedicated to the dynamic computations of the DSL, based on the semantic algebras. To facilitate this process, the dynamic parts of the semantic context need to be grouped in a unique semantic argument which prefigures the basic entities of the abstract machine (*e.g.*, registers).

Because all of the dynamic context is passed as a unique argument to the valuation functions, the operations in the semantic algebras no longer need to be passed to the dynamic semantic arguments separately; they can be transformed so as to get these values from a unique argument.

Working example. The valuation functions shown in Figure 4 already group the dynamic semantic arguments passed to the valuation functions into a unique semantic argument, namely *DynamicState*. These arguments naturally correspond to registers of the abstract machine. We group them in the domain $AbsMachState$.

In addition, consider the *new-msg* operator. It returns a fresh, new message whose fields are later assigned dynamically. However, from an operational viewpoint, only two messages may exist at any time: the current inbound message and a message being composed (two new messages cannot be composed at the same time). To make globalization more explicit, we dedicate an extra register of the abstract machine for the message being currently composed. The operator *new-msg* : *Message* thus becomes the abstract machine instruction new-msg : $AbsMachState \rightarrow AbsMachState$ which operates indirectly on this new register.

In making operators like *set-field* and *get-field* implicitly access the dynamic registers, an ambiguity has appeared because we now have two registers for messages. To make the message register explicit, we denote $\textsf{get-field}_i$ the instruction that accesses the inbound message and $\textsf{get-field}_c$ the instruction that accesses the message being composed. In an actual implementation, this may be modeled as an argument to the instructions (*e.g.*, a pointer to the actual message).

The resulting semantic definition based on the abstract machine is given in Figure 5.

7 Implementation

The implementation of a DSL can be derived from the implementation of its valuation function and an implementation of the corresponding abstract machine.

$$\mathcal{C} : \mathit{Command} \to \mathit{StaticState} \to \mathit{AbsMachState} \to \mathit{AbsMachState} \quad \text{where}$$
$$\mathit{StaticState} = \mathit{FolderHierarchy} \times \mathit{UserName}$$
$$\mathit{AbsMachState} = \mathit{FolderFiles} \times \mathit{OutStream} \times \mathit{CmdStream} \times \mathit{Date} \times \mathit{Message}_i \times \mathit{Message}_c$$

$$\mathcal{C}\ [\![C_1\ \texttt{;}\ C_2]\!]\ \rho = (\mathcal{C}\ [\![C_2]\!]\ \rho)\ \circ\ (\mathcal{C}\ [\![C_1]\!]\ \rho)$$

$$\mathcal{C}\ [\![\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\ \rho = \mathsf{cond}\ (\mathcal{B}\ [\![B]\!])\ (\mathcal{C}\ [\![C_1]\!]\ \rho)\ (\mathcal{C}\ [\![C_2]\!]\ \rho)$$

$$\mathcal{C}\ [\![\texttt{skip}]\!]\ \rho = \mathsf{no\text{-}op}$$

$$\begin{array}{l}
\mathcal{C}\ [\![\texttt{copy}\ F]\!]\ \rho\ \sigma = \\
\quad \text{let}\ \nu = \mathsf{get\text{-}filename}\ (\mathcal{F}\ [\![F]\!])\ \rho_{\mathsf{folder\text{-}hierarchy}} \\
\quad \text{in}\ ((\mathsf{write\text{-}folder}\ \nu) \circ \\
\qquad (\mathsf{add\text{-}msg}) \circ \\
\qquad (\mathsf{set\text{-}field}_i\ \text{“}\texttt{Delivery-Date}\text{”}\ \sigma_{\mathsf{date}}) \circ \\
\qquad (\mathsf{read\text{-}folder}\ \nu))\ \sigma
\end{array}$$

$$\begin{array}{l}
\mathcal{C}\ [\![\texttt{forward}\ S]\!]\ \rho\ \sigma = \\
\quad ((\mathsf{send\text{-}msg}) \circ \\
\quad (\mathsf{set\text{-}field}_c\ \text{“}\texttt{Resent-by}\text{”}\ (\mathsf{concat}\ \rho_{\mathsf{user\text{-}name}}\ (\mathsf{get\text{-}field}_i\ \text{“}\texttt{Resent-by}\text{”}\ \sigma))) \circ \\
\quad (\mathsf{set\text{-}field}_c\ \text{“}\texttt{Subject}\text{”}\ (\mathsf{concat}\ \text{“}\texttt{Fwd: }\text{”}\ (\mathsf{get\text{-}field}_i\ \text{“}\texttt{Subject}\text{”}\ \sigma))) \circ \\
\quad (\mathsf{set\text{-}body}_c\ (\mathsf{msg\text{-}to\text{-}string}_i\ \sigma)) \circ \\
\quad (\mathsf{set\text{-}field}_c\ \text{“}\texttt{From}\text{”}\ \rho_{\mathsf{user\text{-}name}}) \circ \\
\quad (\mathsf{set\text{-}field}_c\ \text{“}\texttt{To}\text{”}\ (\mathcal{S}\ [\![S]\!])) \circ \\
\quad (\mathsf{set\text{-}field}_c\ \text{“}\texttt{Date}\text{”}\ \sigma_{\mathsf{date}}) \circ \\
\quad (\mathsf{new\text{-}msg}_c))\ \sigma
\end{array}$$

$$\begin{array}{l}
\mathcal{C}\ [\![\texttt{reply}\ S_1]\!]\ \rho\ \sigma = \\
\quad ((\mathsf{send\text{-}msg}) \circ \\
\quad (\mathsf{set\text{-}field}_c\ \text{“}\texttt{Subject}\text{”}\ (\mathsf{concat}\ \text{“}\texttt{Re: }\text{”}\ (\mathsf{get\text{-}field}_i\ \text{“}\texttt{Subject}\text{”}\ \sigma))) \circ \\
\quad (\mathsf{set\text{-}body}_c\ (\mathcal{S}\ [\![S_1]\!])) \circ \\
\quad (\mathsf{set\text{-}field}_c\ \text{“}\texttt{From}\text{”}\ \rho_{\mathsf{user\text{-}name}}) \circ \\
\quad (\mathsf{set\text{-}field}_c\ \text{“}\texttt{To}\text{”}\ (\mathsf{get\text{-}field}_i\ \text{“}\texttt{From}\text{”}\ \sigma)) \circ \\
\quad (\mathsf{set\text{-}field}_c\ \text{“}\texttt{Date}\text{”}\ \sigma_{\mathsf{date}}) \circ \\
\quad (\mathsf{new\text{-}msg}_c))\ \sigma
\end{array}$$

$$\mathcal{C}\ [\![\texttt{pipe}\ S]\!]\ \rho = \mathsf{pipe\text{-}msg}\ (\mathcal{S}\ [\![S]\!])$$

$$\mathcal{B} : \mathit{BoolExpr} \to \mathit{AbsMachState} \to \mathit{AbsMachState}$$

$$\mathcal{B}\ [\![\texttt{match}\ S_1\ S_2]\!]\ \sigma = \mathsf{match}\ (\mathsf{get\text{-}field}_i\ (\mathcal{S}\ [\![S_1]\!])\ \sigma)\ (\mathcal{S}\ [\![S_2]\!])$$

Fig. 5. Abstract-machine-based semantic definition of MAILSH

Like GPLs, DSLs can either be implemented by an interpreter or a compiler. The abstract mahcine provides a portable layer.

7.1 Interpretation

The interpretation is usually the easiest implementation approach because it processes a program in the presence of its data, and thus directly produces an answer. In contrast, a compiler produces a program which, when executed, produces a result. Thus, the compiler approach introduces an indirection which makes it more difficult to develop. Another advantage of the interpretation approach is that interpreters can often be derived from the denotational definition by directly translating the specification into a functional program.

The interpretation approach is also well known for its flexibility. For example, there are existing techniques to extend interpreters (*e.g.*, based on monads [22, 32, 37]) without corrupting the semantics. More generally, interpretation allows languages to be prototyped rapidly and thus language design can be very reactive. This feature is particularly important in the context of a DSL; given that needs in the domain may evolve over time, so should the DSL. The obvious limitation of the interpretation approach is inefficiency. Depending on the language, interpretation has been commonly cited to be one order of magnitude slower than compiled code [30].

7.2 Compilation

From a software engineering viewpoint, a DSL compiler can be seen as an application generator in that it processes a specification to generate an application. Traditional compilation could be applied to a DSL; that is, native code could be directly produced from a DSL program. Developing a compiler which generates efficient code should not require more effort than for a GPL, considering the restricted nature of a DSL.

Another compilation strategy consists of producing abstract machine instructions from a DSL program. In doing so, the staged semantics is exploited to allow more flexibility in the implementation of the abstract machine.

7.3 Abstract Machine Implementation

As for efficiency, the abstract machine layer should cause a negligible overhead given that each instruction often captures substantial dynamic computations. Therefore, if there exist efficient compilers for the implementation language of the abstract machine, little (if any) overhead should be incurred compared to natively-compiled programs.

Depending on the implementation language which *glues* the abstract machine instructions, the valuation functions may however need linearization. If the implementation language has expressions, the abstract machine code may stay structured: the reason is that the implementation language compiler would linearize them anyways. In our example, these are the expressions involving msg-to-string, concat, get-field, etc. On the contrary, linearizing these instructions further would have been useless, at best, and an obstacle for optimizing compilation, at worst. Indeed, higher-level machine instructions may expose more optimization opportunities than instructions where early operational choices have been made. If the implementation language is flat (*e.g.*, assembly, JVM), then linearization is necessary. However, linearization does not go beyond state transition boundaries.

7.4 The Abstract Machine as an API

Although only a single implementation of the valuation functions is typically needed, there might be several implementations of the same abstract machine, to account for different operational contexts.

Working example. To illustrate the flexibility offered by a staged implementation, let us examine the MAILSH folders. There are several common implementations of a folder, either as a single file being the (formatted) concatenation of messages (*e.g.*, Netscape or GNU Emacs) or as a directory containing one file per message (*e.g.*, `exmh`). We abstracted over these implementation choices by introducing the domains *FileName* and *FolderFiles*.

8 Partial Evaluation

In the previous section, two separate implementation approaches were presented, namely, interpretation and compilation. In this section we propose a third approach which allows compilation to be achieved from an interpreter. This approach relies on partial evaluation [6, 16, 17]. It consists of developing an interpreter based on the staged semantics of the DSL. Then, a partial evaluator is applied to the interpreter and a given DSL program to process the static semantics. That is, it performs the static computations of the interpreter and produces code for the dynamic computations, as a compiler would do.

Partial evaluation has traditionally been used to specialize an interpreter by removing the interpretation layers [10, 18]. More generally, it has been shown to successfully optimize the implementation of various software architectures [7, 23]. In the context of DSLs, partial evaluation has been used to successfully optimize DSL interpreters, as demonstrated by GAL, a language to specify device drivers for PC graphics card. Thibault *et al.* have reported that the GAL interpreter can be specialized with respect to a driver specification (known at compile time) to yield an implementation as efficient as an equivalent, hand-written device driver [35].

In addition, partial evaluation has been successfully used to specialize interpreters at *run time*, *i.e.*, with respect to a DSL program not known until run time. This work has been done in the context of PLAN-P, a DSL for active networks [34]. When the PLAN-P interpreter is specialized at run time with respect to a PLAN-P program, the resulting code incurs no overhead in overall system performance in comparison with hand-written C code. Furthermore, in comparison with Java, another mobile code approach, the specialized program

```
cond (match (get-field_i "Subject") "DSL")(
  new-msg;
  set-field_c "Date" (date);
  set-field_c "To" "jake";
  set-field_c "From" "bob";
  set-body_c  (msg-to-string_i);
  set-field_c "Subject" (concat "Fwd: " (get-field_i "Subject"));
  set-field_c "Resent-by" (concat "bob" (get-field_i "Resent-by"));
  send-msg;
  read-folder "/home/bob/Mail/Research/Lang/DSL";
  set-field_c "Delivery-Date" (date);
  write-folder "/home/bob/Mail/Research/Lang/DSL"
)(
  cond (match (get-field_i "From") "hotmail.com")(
    new-msg;
    set-field_c "Date" (date);
    set-field_c "To" (get-field_i "From");
    set-field_c "From" "bob";
    set-body_c  "Leave me alone!";
    set-field_c "Subject" (concat "Re: " (get-field_i "Subject"));
    send-msg;
  )(
    cond (match (get-field_i "Subject") "seminar")(
      pipe-msg "agenda --stdin"
    )(
      no-op)))
```

Fig. 6. Implementation of a MAILSH program example

is twice as fast as an equivalent Java program compiled with an optimizing off-line byte-code compiler. In effect, run-time specializing interpreters achieve the same functionality as a *Just-In-Time* compiler for the price of an interpreter. Moreover, unlike specialized GPL interpreters, which compete with optimizing compilers producing fine-grained, low-level operations, specialized DSL interpreters can yield domain-specific, coarse-grained operations where the need for efficiency often resides.

Notice that the specialization of both DSL interpreters (GAL and PLAN-P) were done using Tempo [8, 9], a partial evaluator for the C language developed by the Compose group (`http://www.irisa.fr/compose/tempo`).

Working example. Recall the example of a MAILSH program presented in Figure 3. We have taken its denotation and performed all the reductions made possible by the availability of both the program and folder hierarchy. Unlike GAL and PLAN-P, this was done by hand; MAILSH has not been implemented. The resulting term is presented in Figure 6. To illustrate the globalization phase, the dynamic state is eliminated from the reduced denotation; the composition of the state-transforming instructions is noted with a semicolon. As can be noticed, the result is quite close to an imperative program. This representation could be transformed into a very efficient C program for example.

9 Conclusion

DSLs have been successfully used to address software engineering concerns in specific application domains. Yet, methodologies for language development have been focusing on GPLs, designed to be universal. In this paper, we have proposed an approach aimed at bridging the gap between these two perspectives. This approach is a complete software development process starting from the identification of the need for a DSL to its efficient implementation. It uses the denotational framework to formalize the basic components of a DSL. The semantics definition is structured so as to stage design decisions and to smoothly integrate implementation concerns. When implemented as an interpreter, partial evaluation is proposed as an optimization technique to remove the performance overhead. Our methodology builds on two successful developments of DSLs: GAL, a language to specify device drivers for graphics cards, and PLAN-P, a language to program routers.

Beyond a methodology to develop DSLs, we are now studying an approach to allowing one to assemble a DSL from parameterized building blocks. This work stems from the fact that, although specific to a domain, a DSL often includes common functionalities which could correspond to generic components. Providing these components in a DSL development environment could facilitate the work for non-experts in programming languages to develop their own DSL. A related topic involves the definition of properties about these components such that they could be safely composed when defining a new DSL.

Another avenue of research consists of exploring structuring techniques for the DSL definition to enable the derivation of DSL program analyzers. A departure point for this study would include factorized semantics as proposed by Jones and Nielson [15].

Finally, the methodology needs to be further validated by more applications. We plan on investigating other families of problems to develop new DSLs. To do so, we are actively studying networking where various DSL candidates have been identified (*e.g.,* Web caching).

Acknowledgments

A substantial amount of the research reported in this paper builds on work done by the authors with Scott Thibault on DSLs. Another source of inspiration comes from work done by the first author with Olivier Danvy in the early nineties.

We thank Julia Lawall and Scott Thibault for thoughtful comments on earlier versions of this paper, as well as the Compose group for stimulating discussions.

References

1. B.R.T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language describing financial products. In *IEEE Workshop on Formal Methods Application in Software Engineering*, pages 6–13, April 1995.
2. E. Bjarnason. Applab: a laboratory for application languages. In L. Bendix, K. Nørmark, and K Østerby, editors, *Nordic Workshop on Programming Environment Research, Aalborg*. Technical Report R-96-2019, Aalborg University, May 1996.
3. G. Booch. *Software Components with Ada*. Benjamin Cummings, 1987.
4. Satish Chandra and James Larus. Experience with a language for writing coherence protocols. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.
5. J. Graig Cleaveland. Building application generators. *IEEE Software*, July 1988.
6. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.
7. C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Partial evaluation for software engineering. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.
8. C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.
9. C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.
10. C. Consel and Danvy O. Static and dynamic semantics processing. In *Conference Record of the Eighteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, Orlando, FL, USA, January 1991. ACM Press.
11. D. Cuka and D. Weiss. Engineering domains: Executable commands as an example. In *Proc. Fifth Internationbal Conference on Software Reuse*, June 1998.
12. Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.
13. N.K. Gupta, L. J. Jagadeesan, E. E. Koutsofios, and D. M. Weiss. Auditdraw: Generating audits the fast way. In *Proceedings of the Third IEEE Symposium on Requirements Engineering*, pages 188–197, January 1997.
14. N. D. Jones, editor. *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
15. N. D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program ana lysis. Technical report, University of Copenhagen and Aarhus University, Copenhagen, Denmark, 1990.
16. N.D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, sep 1996.
17. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

18. N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
19. Samuel Kamin and David Hyatt. A special-purpose language for picture-drawing. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.
20. R. Kieburtz, L. McKinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th IEEE International Conference on Software Engineering ICSE-18*, pages 542–553, 1996.
21. D. Ladd and C. Ramming. Two application languages in software production. In *USENIX Symposium on Very High Level Languages*, New Mexico, October 1994.
22. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 333–343. ACM, January 1995.
23. R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183–192, Lake Tahoe, Nevada, November 1997. IEEE Computer Society.
24. R. McCain. Reusable software component construction: A product-oriented paradigm. In *Proceedings of the 5th AiAA/ACM/NASA/IEEE Computers in Aerospace Conference*, Long Beach, California, October 1985.
25. James Neighbors. *Software Construction Using Components*. PhD thesis, University of California, Irvine, 1980.
26. D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2:1–9, mar 1976.
27. G. D. Plotkin. *A Structural Approach To Operational Semantics*. University of Aarhus, Aarhus, Denmark, 1981.
28. Rubén Prieto-Díaz. Domain analysis: An introduction. *Software Engineering Notes*, 15(2), April 1990.
29. C. Pu, A. Black, C. Cowan, J. Walpole, and C. Consel. Microlanguages for operating system specialization. In *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997. Computer Science Technical Report, University of Illinois at Urbana-Champaign.
30. T. Romer, D. Lee, G. Voelker, A. Wolman, W. Wong, J. Baer, B. Bershad, and H. Levy. The structure and performance of interpreters. In *Proceedings of 7th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, October 1996.
31. D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
32. Guy L. Steele. Building interpreters by composing monads. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*. ACM Press, 1994.
33. S. Thibault and C. Consel. A framework of application generator design. In M. Harandi, editor, *Proceedings of the Symposium on Software Reusability*, pages 131–135, Boston, Massachusetts, USA, May 1997. Software Engineering Notes, 22(3).
34. S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.

35. S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device drivers: from design to implementation. In *Conference on Domain Specific Languages*, pages 11–26, Santa Barbara, CA, October 1997. Usenix.
36. Scott Thibault, Renaud Marlet, and Charles Consel. A domain-specific language for video device driver: from design to implementation. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.
37. P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 1–14, Albuquerque, New Mexico, USA, January 1992. ACM Press.
38. Bruce W. Weide and William F. Ogden. Recasting algorithms to encourge reuse. *IEEE Software*, 11(5), September 1994.
39. D.M. Weiss. Family-oriented abstraction specification and translation: the fast process. In *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS), Gaithersburg, Maryland*, pages 14–22. IEEE Press, Piscataway, NJ, 1996.