A Module System for Domain-Specific Languages

ETHAN K. JACKSON
Research In Software Engineering (RISE)
Microsoft Research, Redmond, WA, USA 98052
(e-mail: ejackson@microsoft.com)

submitted ; revised ; accepted

Abstract

Domain-specific languages (DSLs) are routinely created to simplify difficult or specialized programming tasks. They expose useful abstractions and design patterns in the form of language constructs, provide static semantics to eagerly detect misuse of these constructs, and dynamic semantics to completely define how language constructs interact. However, implementing and composing DSLs is a non-trivial task, and there is a lack of tools and techniques.

We address this problem by presenting a complete module system over LP for DSL construction, reuse, and composition. LP is already useful for DSL design, because it supports executable language specifications using notations familiar to language designers. We extend LP with a module system that is simple (with a few concepts), succinct (for key DSL specification scenarios), and composable (on the level of languages, compilers, and programs). These design choices reflect our use of LP for industrial DSL design. Our module system has been implemented in the FORMULA language, and was used to build key Windows 8 device drivers via DSLs. Though we present our module system as it actually appears in our FORMULA language, our emphasis is on concepts adaptable to other LP languages.

KEYWORDS: module systems, domain-specific languages, logic programming

1 Introduction

Domain-specific languages (DSLs) are routinely created to simplify difficult or specialized programming tasks (Cartey et al. 2012; Sangiovanni-Vincentelli et al. 2009). They expose useful abstractions and design patterns in the form of language constructs. Unlike libraries, which also expose abstractions, DSLs have static semantics to eagerly detect misuse of constructs and dynamic semantics to completely define how language constructs interact (independently of implementation). However, these advantages come with at least two disadvantages: (1) Language design and implementation is challenging, requiring formal specifications, compilers, and debuggers. (2) Composing and reusing DSLs is non-trivial, whereas, on the surface, composing and reusing libraries is as simple as importing them and calling their APIs.

Logic programming (LP) is useful for DSL design and implementation (Gurevich 2012; Alvaro et al. 2010). Traditionally, programming languages have been specified with a combination of algebraic data types (for ASTs) (Hudak 1996), rules of inference (for typing and static semantics) (Cardelli 1997), and abstract transition systems (for dynamic semantics) (Börger 2005). These specification styles are closely related to logic
### Modules

| Kind    | Purpose                                                                 | Section |
|---------|-------------------------------------------------------------------------|---------|
| Domain  | Describes DSL syntax, type judgments, and static semantics using algebraic data types (ADTs) and LP. | 2.1, 2.2 |
| Model   | Represents a DSL program w.r.t. a domain as a set of ground facts.       | 2.3     |
| Transform | A function from models to models, such as a compiler or transition system. Defined using ADTs and LP. | 4.2     |

### Language Features

| Feature            | Purpose                                                                                                                                                                                                 | Section |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|
| Contracts          | Conforms clauses specify DSL static semantics. Requires and ensures clauses specify transform behavior.                                                                                               | 2.2, 4.4|
| Symbolic constants | Module-level constants for labeling common sub-expressions and naming value-level transform parameters.                                                                                               | 2.3     |
| Inferred rewrites  | Compiler-inferred term rewrites that can replace boilerplate recursive rules in transforms.                                                                                                          | 1.3     |

### Composition Operators

| Feature         | Purpose                                                                                                                                                                                                 | Section |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|
| Extends, Includes | Compose DSL syntax and static semantics (for domains) and DSL programs (for models); extends is conjunctive for static semantics.                                                                   | 3.2, 5.3|
| Renaming (::)   | Generates new modules by systematically renaming data constructors. Used to create product DSLs and define transforms.                                                                                 | 4.1     |
| Sequential composition | Creates larger transforms by sequential composition. For instance, a compiler can be decomposed into a sequence of small transforms.                                                                 | 4.4     |

Table 1. Overview of module system.

programs, and rephrasing them as proper logic programs yields formal specifications of DSL semantics that are also language implementations. However, existing LP systems have much less support for composing and reusing DSL specifications. Most systems provide modules for defining and exporting predicates, but this is a low-level form of composition from the perspective of DSLs and compilers (Hammerlé and Fages 2006).

In this paper we present a complete module system over LP for DSL construction, reuse, and composition. Table 1 gives an overview of our module system and the structure of this paper. It has been designed to be simple (with a few concepts), succinct (for key DSL specification scenarios), and composable (on the level of languages, compilers, and programs). These design choices reflect our use of LP for industrial DSL design. Our module system has been implemented in the FORMULA language (http://formula.codeplex.com), and has been used to build DSLs for modeling cyber-physical systems (Simko et al. 2013), programming Windows device drivers (Desai et al. 2013), and specifying resource allocation problems (Jackson et al. 2010), to name a few. Windows 8.0 ships with core components that were built using DSLs specified with FORMULA (e.g. the USB 3.0 stack).

We demonstrate our module system by modularly developing a finite state machine DSL with a small action language. Our examples are as they appear in our FORMULA lan-
Fig. 1. Domain module defining a class of non-deterministic finite state machines.

Domains, Models, and Conformance

Domains specify DSL syntaxes and static semantics, whereas models represent DSL programs. There is a conformance relationship between domains and models: a model conforms to its domain if it satisfies the domain’s static semantics. Domains encapsulate algebraic data type (ADT) definitions, rules of inference, and static semantics. Figure 1 shows the NonDetFSM domain for a class of non-deterministic finite state machines. Lines 3 - 6 are ADT definitions for the syntax of the DSL. Lines 8 and 9 define the syntax and semantics of the Reach judgment, which judges the reachable machine states. Finally, lines 11 - 17 give the static semantics.

2.1 Syntaxes and Algebraic Data Types

ADTs are standard for representing the syntaxes of DSLs and judgments, and our module system makes heavy use of them. Our ADT definitions come in two forms. The first form defines a data constructor $F$:

$$F ::= \text{new} \ (\text{arg}_1 : \cdot T_1, \ldots, \text{arg}_n : \cdot T_n).$$

where $[e]$ is an optional expression. This form introduces an $n$-ary data constructor $F()$ and an identically named type $F$. The type $F$ denotes all values obtained by applying $F()$ to arguments $a_1, \ldots, a_n$ belonging to types $T_1, \ldots, T_n$. For instance, the $\text{State}$ type denotes the set of all values constructed by applying $\text{State}()$ to an integral value. Applying $\text{State}()$ to a non-integral value creates a badly-typed value. The FORMULA compiler
E. K. Jackson statically guarantees that badly-typed values can never be created (Jackson et al. 2012). The arguments of data constructors can optionally be named, e.g. the first argument of State() is called id. Data constructors marked with the new keyword are used for DSL syntax, whereas unmarked constructors are used for the syntax of judgments. Notice that the Reach() constructor is unmarked, and can never appear as part of DSL syntax. The second form of type definition defines a union type and has the form: \( U := T_1 + \ldots + T_n \). Here, \( U \) denotes the mathematical union of the types \( T_1, \ldots, T_n \).

**Related work.** There are many variations of ADTs. Our ADTs are a sub-class of regular types, with the useful properties that type equality is co-NP-complete and type expressions have canonical forms (Jackson et al. 2011). Generalizing to full regular types significantly changes the complexity of static checking, as type equality becomes EXPTIME-complete and types become as expressive as tree automata (Aiken and Murphy 1991).

For instance, let \( s() \) be a unary data constructor standing for the successor function, then regular types can distinguish between even and odd naturals with the productions:

\[
\text{Even} \rightarrow 0. \quad \text{Even} \rightarrow s(\text{Odd}). \quad \text{Odd} \rightarrow s(\text{Even}).
\]

Few existing LP systems implement full regular types (e.g. Ciao (Hermenegildo et al. 2005)). In our experience, a weaker type system is sufficient for capturing syntax, and then more complex properties can be stated directly using LP.

ADTs are also standard in functional programming languages where they are generalized along different dimension compared to regular types (Sulzmann et al. 2006). For example, consider these recursive type definitions for two different kinds of lists in the Haskell language (Heeren et al. 2003):

\[
\text{data ListInt} = \text{ConsInt} \text{ Integer ListInt} \mid \text{NilInt} \\
\text{data ListStr} = \text{ConsStr} \text{ String ListStr} \mid \text{NilStr}
\]

In Haskell each definition requires a distinct \textit{Nil} constructor (i.e. \textit{NilInt} and \textit{NilStr}), otherwise the type of \textit{Nil} would be both \textit{ListInt} and \textit{ListStr}. In our experience, this restriction is too strong, because it hinders composition of DSL syntaxes. In FORMULA these two definitions can share the same \textit{Nil} constructor as follows:

\[
\text{ConsInt} ::= (\text{Integer}, \text{ListInt}). \quad \text{ListInt} ::= \text{ConsInt} + \{ \text{Nil} \}.
\]

\[
\text{ConsStr} ::= (\text{String}, \text{ListStr}). \quad \text{ListStr} ::= \text{ConsStr} + \{ \text{Nil} \}.
\]

(The FORMULA syntax \( \{ c_1, \ldots, c_n \} \) denotes a set of constants.) Consequently, FORMULA values do not have unique minimal types. This allows syntactic elements to be mixed more freely in composite languages, but comes at the cost of higher-complexity type equality and sub-type testing.

### 2.2 Judgments and Conformance

LP is used to define rules of inference and static semantics. For example, the reachability judgment (line 10) is a standard recursive rule marking a state as reachable if it can be reached from an initial state by some transitions. To be clear, FORMULA does not support definitions of new program relations. Rules examine and populate an implicit unary program relation \( p() \), and stratification conditions are more fine-grained to account for this. For instance, if \( p() \) were made explicit, then the reachability rule would be written: \( p(\text{Reach}(s)) :- p(\text{Init}(s)); p(\text{Reach}(s')), p(\text{Trans}(s', \_ , s)) \).
Fig. 2. Several FSM programs encapsulated within models.

Static semantics are described through conforms clauses; such a clause has the form: conforms body. Where body is any expression that could appear on the right-hand side of a standard rule. A DSL program conforms to its domain if every conforms clause is provable, i.e., conforms clauses are conjunctive.

The conforms clause in line 11 requires a FSM to have at least one initial state. The remaining clauses encode a convention that states and events are “defined” by introducing them into the program relation p(). Initial states and transitions should only refer to these “defined” values. For instance, it is expected that: \( \forall x. p(\text{Init}(\text{State}(x))) \Rightarrow p(\text{State}(x)). \)

The other conforms clauses check these properties using double-negations. Line 13 is read as: “There should not exist an \( x \) where \( p(\text{Init}(\text{State}(x))) \) and not \( p(\text{State}(x)) \).” The syntax \{head | body\} stands for the set of values produced by the rule head :- body, and the no operator tests if this set is empty. This syntactic sugar allows negations to be nested within a FORMULA rule, though negations must still be stratified. Finally, the syntax \( x \in T \) is shorthand for \( p(x), x : T \), meaning \( x \) holds in \( p() \) and has type \( T \). This syntax allows variables to range over elements of \( p() \) based on their type and helps to avoid long patterns of the form \( f(\ldots, x, \ldots) \).

Related work. A number of LP languages support integrity constraints (ICs) through rules of the form false :- body. A program able to prove the body of an integrity constraint is inconsistent. One could easily envision replacing conforms clauses with ICs. However, there is an important difference. In our approach, the inability to prove conformance is not a logical inconsistency, and this fact allows for more flexible forms of domain composition. The semantics of ICs is rigid and so composing two sets of ICs means all ICs must hold in the composition or the program is logically inconsistent.

2.3 Models

Models define DSL programs. The contents of a model is a set of well-typed ground facts constructed using the new-kind constructors of the model’s domain. Semantically, a model is a logic program formed by composing its domain with the set of ground facts enumerated in the model. Figure 2 shows several examples of FSM models. The OneStateMach model defines a FSM consisting of a state with ID 1, an event named “Foo”, marks 1 as the initial state, and introduces a self-transition on 1 triggered by “Foo”. Composing these facts with the NonDetFSM domain produces a program where all conforms clauses are provable, and so the model is statically correct. The BadMach
model does not conform, because the initial state (i.e. State(100)) is not an element of p(). It violates the conforms clause in line 13 of Figure 1.

The TwoStateMach model uses symbolic constants, e.g. s1 and eFoo, to reuse program expressions. Symbolic constants are defined in models using the form: c is F(t1, . . . , tn). Such a definition introduces the fact p(F(t1, . . . , tn)) and defines a module-level constant c that evaluates to F(t1, . . . , tn). Symbolic constants enable snippets of programs to be shared, sometimes leading to exponentially succinct models. The order of definitions does not matter, though every symbolic constant must be defined and definitions must be acyclic.

### 3 Domain and Model Composition

#### 3.1 Symbol Tables

Composition is by unioning module definitions. The mechanics of composition depends on symbol definitions and their organization into symbol tables. Table 2 lists the kinds of user-defined symbols along with example introductions. Constructors that may appear in a DSL’s syntax are called new-kind constructors, whereas those only permitted in judgments are called derived-kind constructors. Constants are treated as nullary constructors, and derived-kind constants are introduced by using them on the LHS of rules (and can behave like propositions). A standard idiom for declaring a derived-kind constant q is by the tautology q :- q. Symbols can only have one kind. For instance, in the rule x :- f(x) the symbol x appears both as a variable and a derived-kind constant, which causes a compile-time error. Built-in symbols such as 1 and “foo” are new-kind constants, and built-in data types such as Integer and String are union types.

Every module has a symbol table recording all definitions. Let S be the (infinite) set of all possible symbols, $\mathcal{K} \overset{\text{def}}{=} \{\eta, \delta, \mu, \nu, \sigma\}$ be the set of symbol kinds, and $\mathcal{T}$ be the set of all terms that can be constructed from S. A symbol table table is a partial function from symbols to triples, table : $\mathcal{S} \rightarrow \mathcal{K} \times \mathcal{N} \times 2^\mathcal{T}$. If table(s) = (k, n, T) then symbol s is defined in table to have kind k, arity n, and T is the set of well-typed terms that s can represent. Here are some examples:

\[
\begin{align*}
table(1) &= (\eta, 0, \{1\}), \\
table(x) &= (\nu, 0, T), \\
table(Integer) &= (\mu, 0, \{\ldots, -1, 0, 1, \ldots\}), \\
table(State) &= (\eta, 1, \{\ldots, State(-1), State(0), State(1), \ldots\}).
\end{align*}
\]

A nullary constructor can only construct a single well-typed term. For non-nullary con-
Theory and Practice of Logic Programming

| Look-up operation | Result | Explanation |
|-------------------|--------|-------------|
| lookup(table, ε, f) | f      | Symbol with shortest qualifier. |
| lookup(table, A, A.f) | A.A.f  | Symbol with shortest qualifier. |
| lookup(table, A, A.A.f) | ⊥      | No such symbol. |
| lookup(table, ε, B.g) | ⊥      | Ambiguous, no unique embedding. |
| lookup(table, ε, B.C.g) | A.B.C.g | Only compatible symbol. |
| lookup(table, B, C.g) | ⊥      | No such symbol. |

Table 3. Results of `lookup()` on a table with the symbols: f, A.f, A.A.f, A.B.C.g, A.C.B.g.

Constructors and union types, $T$ is the set of all well-typed terms obeying their type definitions. Variables can be substituted for any term. For symbolic constants, $T$ estimates the evaluation of the constants. The composition of symbol tables $\text{table}_1$ and $\text{table}_2$ is defined as follows. For each $s \in \text{dom } \text{table}_1 \cup \text{dom } \text{table}_2$:

$$(\text{table}_1 \oplus \text{table}_2)(s) \rightarrow \begin{cases} \text{table}_1(s) & \text{if } s \in \text{dom } \text{table}_1 - \text{dom } \text{table}_2, \\ \text{table}_2(s) & \text{if } s \in \text{dom } \text{table}_2 - \text{dom } \text{table}_1, \\ (k, n, T) & \text{if } (k, n, T) = \text{table}_1(s) = \text{table}_2(s), \\ ⊥ & \text{otherwise.} \end{cases}$$

The composition of tables is legal if no symbol is mapped to ⊥. This definition allows the same type to be defined in syntactically different ways in several modules. As long as these definitions are semantically equivalent, then the modules can be composed. This flexibility is implemented in `FORMULA`.

3.1.1 Qualified Symbols and Name Resolution

Returning to the set of symbols $S$, we add a free associative operator ‘.’ for constructing qualified symbols. If $x, y \in S$, then $x.y \in S$ and $(x.y).z = x.(y.z)$. Some examples of qualified symbols are:

$$\delta \text{ MyModule.conforms}, \ (\eta) \text{ Left.State}, \ (\sigma) \text{ MyModel.eFoo}, \ (\eta) \text{ In.Left.Trans}$$

Compared to other languages, our module system makes extensive use of qualifiers, so it is important to provide a succinct name resolution strategy. Our strategy is based on qualifier embeddings. Let $\vec{q}.s$ be a sequence of atomic symbols $q_1, \ldots, q_n$ qualifying the atomic symbol $s$. Let $\epsilon$ be the empty sequence of qualifiers, e.g. $\epsilon.s = s$. A sequence $\vec{p}$ is embedded in $\vec{q}$, written $\vec{p} \sqsubseteq \vec{q}$, if $\vec{p} = \epsilon$ or there is a monotone function $\iota$ from $\vec{p}$-indices to $\vec{q}$-indices such that $p_1 = q_{\iota(1)}, \ldots, p_{|\vec{p}|} = q_{\iota(|\vec{p}|)}$. For example: $b_1.b_2 \sqsubseteq a_1.b_1.a_2.b_2$. The function $\text{lookup}(\text{table}, \vec{r}, \vec{p}.s)$ returns the shortest symbol from $\text{table}$ that begins with $\vec{r}$ and embeds $\vec{p}$. Specifically, $\text{lookup}(\text{table}, \vec{r}, \vec{p}.s) \rightarrow \vec{r}\vec{q}.s$ if $\vec{p} \sqsubseteq \vec{q}$ and $\vec{r}\vec{q}.s \in \text{dom } \text{table}$. And, for every other $\vec{q}' \neq \vec{q}$ if $\vec{r}\vec{q}'.s \in \text{dom } \text{table}$ then $|\vec{q}'| > |\vec{q}|$. If no such $\vec{q}$ exists then $\text{lookup}(\text{table}, \vec{r}, \vec{p}.s, \vec{r}) \rightarrow \perp$. Our $\text{lookup()}$ operation searches for symbols in a more general manner than found in other languages. Consequentially, qualified symbols are mostly invisible to the user, even though our composition operators make heavy use of qualifiers. When qualifiers cannot be avoided, the most meaningful qualifiers can be used to disambiguate the symbol. Table 3 illustrates the results of $\text{lookup()}$ on a sample table.
Fig. 3. Deterministic FSMs with an action language defined via domain composition.

3.2 Domain Composition

A domain $D \triangleq \langle \text{table}, \text{rules} \rangle$ consists of a symbol table and set of rules. As a design decision, the derived-kind constants of a domain $D$ are protected by the qualifier $D$. For instance, the rule \( q :\rightarrow q \) actually introduces a symbol $D.q$ into table. Every domain has a constant $D.conforms$ that is provable if and only if all conforms clauses are provable. The ‘,’ operation merges two domains: $D_1, D_2 \triangleq \langle \text{table}_1 \oplus \text{table}_2, \text{rules}_1 \cup \text{rules}_2 \rangle$. The result is well-defined if $\text{table}_1 \oplus \text{table}_2$ does not map a symbol to $\bot$, and if the $\text{rules}_1 \cup \text{rules}_2$ obey stratification conditions. The includes and extends operators allow a domain $D'$ to import a set of domains.

\begin{verbatim}
  domain D' includes D_1, ..., D_n { ... }.
  domain D' extends D_1, ..., D_n { ... }.
\end{verbatim}

In both cases, $D'$ is treated as if it contains the merged domain $D_1, ..., D_n$. The extends operation adds the additional conforms clause to $D'$:

\begin{verbatim}
  conforms D_1.conforms, ..., D_n.conforms.
\end{verbatim}

The extends operation gives a standard mechanism to conjunct conforms clauses, whereas the includes operation allows the user to define their own composite conformance using the derived-kind constants $D_1.conforms, ..., D_n.conforms$.

Figure 3 uses the merge and extends operators to define a composite DSL of deterministic FSMs with an action language. The action language (see Appendix A) introduces state variables and actions, which are sequential programs that update state variables. The static semantics of the Actions domain includes rules for static type-checking of action bodies. The composite DSL introduces an ActMap constructor relating a state to an action that should be executed upon entering that state. The fun keyword is like new, but requires ActMap to be functional (i.e. for all $x$ there is at most one $y$ s.t. $p(\text{ActMap}(x, y))$). Finally, the composite language conjoins additional conforms clauses:

(1) There is at most one initial state. (2) The transition table of every state is deterministic. (3) Every state is reachable. (4) Every action named in ActMap has been declared.

3.3 Model Composition

A model $M \triangleq \langle \text{table}, \text{facts} \rangle$ is a symbol table and set of facts. $M$ inherits the symbol table of its domain and contains a definition for each symbolic constant $c$ under the fully qualified name $M.c$. Models compose similarly to domains, by composing their
symbol tables and unioning fact sets. The \textit{CntrMach} model in Figure 4 describes a counter machine that increments the state variable “X” on every “Foo” event. It is defined by composing the \textit{CntrActions} model with the previous \textit{TwoStateMach} model. \textit{CntrActions} declares an integer variable named “X”, an action named “ZeroX” for setting “X” to zero, and an action named “IncX” for incrementing “X”. \textit{CntrMach} assigns state \textit{s1} to run action “ZeroX” and state \textit{s2} to run action “IncX”.

4 Renaming and Transforms

Transforms are functions from models to models. They are defined with ADTs, strongly-typed logic programs, and a module-level operator called \textit{renaming}.

4.1 Renaming

The renaming operator “::” produces a new module \textit{x::M} that is identical to \textit{M}, except that every occurrence of symbol \textit{s} in \textit{M} becomes \textit{x.s} in \textit{x::M}. As a design decision, variables and new-kind constants retain their original names. The renaming operator allows the easy construction of distinguished copies of modules. For instance, renaming can be used to construct a DSL representing two distinct FSMs running in parallel. Figure 5 shows how this DSL can be constructed. The \textit{ParallelFSMs} domain is a product domain containing two copies of \textit{DetFSMWithActions} under the renamings \textit{left} and \textit{right}. Symmetrically, the renaming operator can be applied to models. The \textit{ParallelCntrs} model contains two renamed copies of \textit{CntrMach}, thereby creating a valid \textit{ParallelFSMs} model. Table \textit{B} lists the symbol table of this composite model.

4.2 Transforms

Transforms use renaming to label their inputs and outputs. A transform has the shape:

\[ \text{transform } T \ (x_1 :: D_1, \ldots, x_m :: D_m) \ \text{returns} \ (y_1 :: E_1, \ldots, y_n :: E_n) \ \{ \ldots \} \]
where $D_i$ and $E_j$ are domains and the labels $x_1, \ldots, x_m, y_1, \ldots, y_n$ are distinct. The transform module $T$ is a composition of its body with the renamed domains in its signature. The signature also indicates that $x_1 :: D_1, \ldots, x_m :: D_m$ are input domains and $y_1 :: E_1, \ldots, y_n :: E_n$ are output domains.

Figure 6 shows a transform that takes an FSM as input and outputs an equivalent FSM where unreachable states and transitions have been pruned away. Notice the use of renamed constructors to distinguish between input and output values. The first rule (line 5) copies all event declarations from the input to the output. The next rule copies only reachable states. The expression $in.\text{Reach}(State(x))$ only requires one qualification, even though $in.\text{Reach}()$ must have arguments of type $in.\text{State}$. This rule is legal because name resolution first uses the qualifier on the outer constructor to resolve the name of an inner constructor (e.g. $\text{lookup}((\text{table}, in, State))$). If this resolution fails, then it is retried without the outer qualifier.

### 4.3 Inferred Term Rewrites

Every initial state is by reachable, so the third rule copies all initial states (line 7). This rule looks trivial, but closer inspection reveals a potential problem. Variable $s$ in the RHS must have type $in.\text{State}$, but $s$ in the LHS must have type $out.\text{State}$. Because these types are disjoint, the compiler should reject this rule as badly typed. Instead, the compiler infers the user’s intent to convert values of the form $in.\text{State}(a)$ to $out.\text{State}(a)$. The rule inferred by the compiler is actually: $\text{out}.\text{Init}(\rho_{in \rightarrow out}(s)) :- in.\text{Init}(s)$. The relabeling function $\rho_{in \rightarrow out}$ rewrites terms by replacing the in qualifier with the out qualifier. Relabeling functions are recursive; they rewrite arbitrarily deep terms. In our experience, inferred rewrites are essential for making renaming operator practical.

Due to space limitations we only define the inference problem, but not the requisite algorithms. Let $x$ be a variable occurring in the RHS of a rule and the set $T_{rhs}$ contain (at least) all the values $x$ takes when the body of the rule is satisfied. Also, suppose $x$ occurs somewhere in the LHS and $\mathcal{T}_{lhs}$ is the set of all values that $x$ is allowed to take in the LHS. Then $x$ is well-typed if there exists a unique relabeling function $\rho_{\overrightarrow{p} \rightarrow \overrightarrow{q}}$ s.t. $\rho_{\overrightarrow{p} \rightarrow \overrightarrow{q}}(T_{rhs}) \subseteq \mathcal{T}_{lhs}$. The relabeling of a new-kind constant is itself. Otherwise:

$$\rho_{\overrightarrow{p} \rightarrow \overrightarrow{q}}(\overrightarrow{p}u.f(t_1, \ldots, t_n)) \overset{\text{def}}{=} \overrightarrow{q}u.f(\rho_{\overrightarrow{p} \rightarrow \overrightarrow{q}}(t_1), \ldots, \rho_{\overrightarrow{p} \rightarrow \overrightarrow{q}}(t_n)).$$
Deciding if such a relabeling exists is non-trivial and requires complex type inference algorithms. These have been implemented in Formula.

4.4 Execution, Contracts and Composition

A transform $T$ is applied to a sequence of models $M_1, \ldots, M_m$ defined over input domains $D_1, \ldots, D_m$. The mechanics of application are as follows: First, $T$ is composed with the renamed models $x_1 :: M_1, \ldots, x_m :: M_m$ and the resulting logic program is executed. Next, the $j^{th}$ output model $N_j$ is constructed by collecting all values $t$ such that $p(t)$ holds and $t = y_j \vec{u}. f(\ldots)$. Finally, the renaming $y_j$ is removed from these values yielding an output model purely in the output domain $E_j$. In symbols:

$$N_j \equiv \{ \rho_{y_j \rightarrow \epsilon}(t) \mid p(t) \text{ and } t = y_j \vec{u}. f(\ldots) \text{ and } \text{kind} (\vec{u}. f) = \eta \}.$$

As usual, only new-kind constructors can appear in output models, hence the extra constraint on the kind of the constructor.

Contracts appear in many programming languages for clearly specifying the intent of methods / functions \cite{Nienaltowski et al. 2009, Barnett and Schulte 2003}. They can be checked at run-time or compile-time tools can attempt to prove their validity. Our contracts allow users to specify required properties of input models, and properties ensured by the transform when all requirements are met. A clause of the form requires body states a required property. A clause of the form ensures body states an ensured property. For example, line 3 of Prune requires the input model to conform to its domain. Line 4 guarantees output conformance when the input conforms. As with conforms clauses, transform contracts are conjunctive, and each transform $T$ has derived-kind constants $T$.requires and $T$.ensures that are provable when all requires and ensures clauses are satisfied. A transform contract is a claim that for every application $T(M_1, \ldots, M_m)$ then $p(T$.requires) ⇒ $p(T$.ensures). Compile-time verification is supported by FORMULA, but outside the scope of this paper.

Finally, transforms can be sequentially composed by listing a series of oriented equations as shown in Figure 7. The PruneAndParallelize composite transform takes two FSMs as inputs, prunes the FSMs, and then combines them into a single ParallelFSMs model. The RHS of each equation is a transform application, and the LHS is a sequence of variables that will be bound to the outputs produced by the application. Such a composite is executed by running each constituent transform in dependency-order, while applying renaming and un-renaming at the boundaries of each step. Transform systems can also be sequentially composed within transform systems.
4.5 Related Work

Prolog specifically (Jouault and Bézivin 2006) and LP generally (Horváth et al. 2010) have been recognized as useful for model transformations. In the former case, Prolog is employed as a behind-the-scenes execution engine, so it was not extended with modules. In the latter case, Prolog-style semantics served as inspiration for the VIATRA2 language, which is not a LP language. However, model transformation languages such as VIATRA2 do commonly support various forms of sequential composition (Bisztray et al. 2009; Boronat et al. 2009), giving further motivation to include it in a module system. LP with constraints has also been used to reason about DSL specifications by translation to LP without extending its module system (Cabot et al. 2007).

DSLs have a long history in the software modeling community. The tools and techniques of the modeling community have been heavily influenced by the Unified Modeling Language (UML), the Object Constraint Language, and the concept of metamodeling. However, the resulting amalgam is difficult to formalize (Boronat and Meseguer 2010) and has a complex and underdeveloped module system (Dingel et al. 2008). By adding a module system for DSLs directly on top of LP, we inherit the well-understood semantics of LP and avoid problems associated with other notations.

5 Discussion and Future Work

In this paper we presented a complete module system on top of LP for the construction, composition, and reuse of DSLs. Our modules, composition operators, and language extensions were designed to be simple yet synergistic. For instance, inferred term rewrites are a general concept and can also be utilized in all modules. Though our running example illustrated a simple compiler, the same mechanism can also be used to specify the elementary steps of transition systems. This use-case is important for dynamic semantics. Our module system evolved over a period of several years, and was informed by both academic and industrial applications of LP for DSL design. For example, we found a contract language to be essential for documenting the intent of modules in an actionable form.

There are still some important ways in which our module system could be extended. One natural desire is a mechanism to make transforms polymorphic on domains. Another is to package the set of transforms defining an abstract transition system into a single module. The contract language for such a package is also an interesting design problem and would most likely fit well with some form of temporal logic. In our opinion more experimentation and use-cases are required before the right combination of extensions becomes clear. We are currently conducting these experiments while applying FORMULA to industrial examples. The examples presented in this paper along with a binary version of FORMULA can be found at [http://research.microsoft.com/en-us/um/redmond/projects/formula/ICLP2014.html](http://research.microsoft.com/en-us/um/redmond/projects/formula/ICLP2014.html). The binary version is compiled for Windows machines. The source code for FORMULA and along with its implementation of this module system can be found at [formula.codeplex.com](http://formula.codeplex.com).
Appendix A  Action Language

domain Actions
{
  /// Declarations
  VarDecl ::= fun (id: String -> type: { BOOL, INT }).
  ActDecl ::= fun (id: String -> action: any Action).

  /// Action language (expressions)
  BoolOp ::= { NOT, AND, OR }.
  IntOp ::= { NEG, ADD, SUB, MUL, DIV }.
  CmpOp ::= { LT, LE, GT, GE, EQ, NEQ }.
  Var ::= new (id: String).
  UnApp ::= new (op: { NEG, NOT }, arg1: any Expr).
  BnApp ::= new (op: { ADD, SUB, MUL, DIV, AND, OR } + CmpOp,
               arg1: any Expr, arg2: any Expr).
  Expr ::= Var + UnApp + BnApp + Boolean + Integer.

  /// Action language (statements)
  Asn ::= new (var: String, expr: any Expr).
  ITE ::= new (cond: any Expr, true: any Action, false: any Action).
  Seq ::= new (act1: any Action, act2: any Action).
  Action ::= Asn + ITE + Seq + { NOP }.

  /// Static typing
  Sub ::= (Action + Expr).
  TypeJudge ::= (Action + Expr, { BOOL, INT, ANY }).
  TypeJudge(e, INT) :-
    Sub(e), e : Boolean;
    TypeJudge(e, INT);
    Sub(e), e = Var(n), VarDecl(n, INT);
    Sub(e), e = UnApp(op, e'), op : IntOp, TypeJudge(e', INT);
    Sub(e), e = Asn(n, e'), VarDecl(n, INT), TypeJudge(e', INT);
    Sub(e), e = BnApp(op, e', e''), op : IntOp,
       TypeJudge(e', INT), TypeJudge(e'', INT).

  TypeJudge(e, BOOL) :-
    Sub(e), e : Boolean;
    Sub(e), e = Var(n), VarDecl(n, BOOL);
    Sub(e), e = UnApp(op, e'), op : BoolOp, TypeJudge(e', BOOL);
    Sub(e), e = Asn(n, e'), VarDecl(n, BOOL), TypeJudge(e', BOOL).
    Sub(e), e = BnApp(op, e', e''), op : BoolOp,
       TypeJudge(e', BOOL), TypeJudge(e'', BOOL);
Sub(e), e = BnApp(op, e', e''), op : CmpOp, 
    TypeJudge(e', t), TypeJudge(e'', t).

TypeJudge(e, ANY) :-
    Sub(e), e = NOP;
    Sub(e), e = Seq(e', e''), TypeJudge(e', _), TypeJudge(e'', _);
    Sub(e), e = ITE(e', e'', e'''), TypeJudge(e', BOOL),
    TypeJudge(e'', _), TypeJudge(e''', _).

    conforms no { e | Sub(e), no { t | TypeJudge(e, t) } }.
### Appendix B Example Symbol Table

| Qualifiers | Name | Kind, Arity | Qualifiers | Name | Kind, Arity |
|------------|------|-------------|------------|------|-------------|
| ADD        | η, 0 | left Init   | ADD        | η, 1 | left η     |
| AND        | η, 0 | left IntOp  | AND        | μ, 0 | right μ    |
| ANY        | η, 0 | left Reach  | ANY        | δ, 1 | right δ    |
| BOOL       | η, 0 | left Seq    | BOOL       | η, 2 | right η    |
| DIV        | η, 0 | left State  | DIV        | η, 1 | right η    |
| EQ         | η, 0 | left Sub    | Eq         | δ, 1 | right δ    |
| FALSE      | η, 0 | left Trans  | FALSE      | η, 3 | right η    |
| GE         | η, 0 | left TypeJudge | GE       | δ, 2 | right δ    |
| GT         | η, 0 | left UnApp  | GT         | η, 2 | right η    |
| INT        | η, 0 | left Var    | INT        | η, 1 | right η    |
| LE         | η, 0 | left VarDecl | LE       | η, 2 | right η    |
| LT         | η, 0 | left Actions | LT       | δ, 0 | right δ    |
| MUL        | η, 0 | left DetFSMWithActions | MUL | δ, 0 | right δ    |
| NEG        | η, 0 | left MachTwoState | NEG | σ, 0 | right σ    |
| NEQ        | η, 0 | left MachTwoState | NEQ | σ, 0 | right σ    |
| NOP        | η, 0 | left MachTwoState | NOP | σ, 0 | right σ    |
| NOT        | η, 0 | left NonDetFSM | NOT | δ, 0 | right δ    |
| OR         | η, 0 | right ActDecl | OR       | η, 2 | right η    |
| SUB        | η, 0 | right ActMap | SUB       | η, 2 | right η    |
| TRUE       | η, 0 | right Action | TRUE      | μ, 0 | right μ    |
| a          | ν, 0 | right Asn   | a          | η, 2 | right η    |
| e          | ν, 0 | right BnApp | e          | η, 3 | right η    |
| e'         | ν, 0 | right BoolOp | e'         | μ, 0 | right μ    |
| e''        | ν, 0 | right CmpOp | e''        | μ, 0 | right μ    |
| i          | ν, 0 | right Expr  | i          | μ, 0 | right μ    |
| n          | ν, 0 | right ITE   | n          | η, 3 | right η    |
| op         | ν, 0 | right Init  | op         | η, 1 | right η    |
| s          | ν, 0 | right IntOp | s          | μ, 0 | right μ    |
| s'         | ν, 0 | right Reach | s'         | δ, 1 | right δ    |
| s''        | ν, 0 | right Seq   | s''        | η, 2 | right η    |
| t          | ν, 0 | right State | t          | η, 1 | right η    |
| ParallelFSMs conform  | δ, 0 | right Sub  | ParallelFSMs conform | δ, 1 | right δ    |
| left ActDecl | η, 2 | right Trans | left ActDecl | η, 3 | right η    |
| left ActMap | η, 2 | right TypeJudge | left ActMap | δ, 2 | right δ    |
| left Action | μ, 2 | right UnApp | left Action | η, 2 | right η    |
| left Asn   | η, 2 | right Var   | left Asn   | η, 1 | right η    |
| left BnApp | η, 3 | right VarDecl | left BnApp | η, 2 | right η    |
| left BoolOp | μ, 0 | right Actions | left BoolOp | δ, 0 | right δ    |
| left CmpOp | μ, 0 | right DetFSMWithActions | left CmpOp | δ, 0 | right δ    |
| left Event | η, 1 | right MachTwoState | left Event | s1     | right s1    |
| left Expr  | μ, 0 | right MachTwoState | left Expr  | s2     | right s2    |
| left ITE   | η, 3 | right NonDetFSM | left ITE   | s0     | right s0    |

Table B.1. Symbol table of composite model *ParallelCntrs* in Figure 5.
References

Aiken, A. and Murphy, B. R. 1991. Implementing Regular Tree Expressions. In FPCA 1991. Springer-Verlag, 427–447.

Alvaro, P., Marczak, W. R., Conway, N., Hellerstein, J. M., Maier, D., and Sears, R. 2010. Dedalus: Datalog in Time and Space. In Datalog. 262–281.

Barnett, M. and Schulte, W. 2003. Runtime verification of .NET contracts. Journal of Systems and Software 65, 3, 199–208.

Bisztay, D., Heckel, R., and Ehrig, H. 2009. Compositionality of Model Transformations. Electr. Notes Theor. Comput. Sci. 236, 5–19.

Börger, E. 2005. Abstract State Machines: a unifying view of models of computation and of system design frameworks. Ann. Pure Appl. Logic 133, 1-3, 149–171.

Boronat, A. and Meseguer, J. 2010. An algebraic semantics for MOF. Formal Asp. Comput. 22, 3-4, 269–296.

Cabot, J., Clarisó, R., and Riera, D. 2007. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In ASE. 547–548.

Cardelli, L. 1997. Type Systems. In The Computer Science and Engineering Handbook. 2208–2236.

Cartey, L., Lyngsø, R., and de Moor, O. 2012. Synthesising graphics card programs from DSLs. In PLDI. 121–132.

Desai, A., Gupta, V., Jackson, E. K., Qadeer, S., Rajamani, S. K., and Zufferey, D. 2013. P: safe asynchronous event-driven programming. In PLDI. 321–332.

Dingel, J., Diskin, Z., and Zito, A. 2008. Understanding and improving UML package merge. Software and System Modeling 7, 4, 443–467.

Gurevich, Y. 2012. Datalog: A Perspective and the Potential. In Datalog. 9–20.

Haemmerlé, R. and Fages, F. 2006. Modules for Prolog Revisited. In ICLP. 41–55.

Heeren, B., Leijen, D., and van IJzendoorn, A. 2003. Helium, for learning Haskell. In Haskell. 62–71.

Hermenegildo, M. V., Puebla, G., Bueno, F., and López-García, P. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). Sci. Comput. Program. 58, 1-2, 115–140.

Horváth, Á., Bergmann, G., Ráth, I., and Varró, D. 2010. Experimental assessment of combining pattern matching strategies with viaatra2. STTT 12, 3-4, 211–230.

Hudak, P. 1996. Building Domain-Specific Embedded Languages. ACM Computing Surveys 28.

Jackson, E. K., Bjørner, N., and Schulte, W. 2011. Canonical Regular Types. In ICLP (Technical Communications). 73–83.

Jackson, E. K., Kang, E., Dahlweid, M., Seifert, D., and Santen, T. 2010. Components, platforms and possibilities: towards generic automation for MDA. In EMSOFT. 39–48.

Jackson, E. K., Schulte, W., and Bjørner, N. 2012. Detecting Specification Errors in Declarative Languages with Constraints. In MoDELS. 399–414.

Jouault, F. and Bézivin, J. 2006. KM3: A DSL for Metamodel Specification. In FMOODS. 171–185.

Nienaltowski, P., Meyer, B., and Ostroff, J. S. 2009. Contracts for concurrency. Formal Asp. Comput. 21, 4, 305–318.

Sangiovanni-Vincentelli, A. L., Shukla, S. K., Sztipanovits, J., Yang, G., and Mathaiikutty, D. 2009. Metamodelling: An Emerging Representation Paradigm for System-Level Design. IEEE Design & Test of Computers 26, 3, 54–69.
Simko, G., Lindecker, D., Levendovszky, T., Neema, S., and Sztipanovits, J. 2013. Specification of Cyber-Physical Components with Formal Semantics - Integration and Composition. In MoDELS. 471–487.

Sulzmann, M., Wazny, J., and Stuckey, P. J. 2006. A Framework for Extended Algebraic Data Types. In FLOPS. 47–64.