Semantics of templates in a compositional framework for building logics

Ingmar Dasseville, Matthias van der Hallen, Gerda Janssens, Marc Denecker
KU Leuven
(*e-mail:* `firstname.lastname@cs.kuleuven.be`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

There is a growing need for abstractions in logic specification languages such as FO(·) and ASP. One technique to achieve these abstractions are templates (sometimes called macros). While the semantics of templates are virtually always described through a syntactical rewriting scheme, we present an alternative view on templates as second order definitions. To extend the existing definition construct of FO(·) to second order, we introduce a powerful compositional framework for defining logics by modular integration of logic constructs specified as pairs of one syntactical and one semantical inductive rule. We use the framework to build a logic of nested second order definitions suitable to express templates. We show that under suitable restrictions, the view of templates as macros is semantically correct and that adding them does not extend the descriptive complexity of the base logic, which is in line with results of existing approaches.

KEYWORDS: compositionality, modularity, templates, macros, semantics, second order logic

1 Introduction

Declarative specification languages have proven to be useful in a variety of applications, however sometimes parts of specifications contain duplicate information. This commonly occurs when different instantiations are needed of an abstract concept. For example, in an application, we may have to assert of multiple relations that they are an equivalence relation, or multiple relations of which we need to define their transitive closure. In most current logics, the constraints (e.g., reflexivity, symmetry and transitivity) need to be reasserted for each relation.

In the early days of programming, imperative programming languages suffered from a similar situation where code duplication was identified as a problem. The first solution proposed to this was the use of macros, where a syntactical replacement was made for every instantiation of the macro. For specification languages, the analog for macros was introduced (e.g. in ASP), most often called *templates*. These allow us to define a concept and instantiating it multiple times, without making the language more computationally complex. Asserting that the two relations `P` and `Q` are equivalence relations could be done using a template `isEqRelation` as follows:

Example 1: This example defines an equivalence relation

```
{isEqRelation(F) ←
  ∀a : F(a,a).
  ∀a,b : F(a,b) ⇔ F(b,a).
  ∀a,b,c : (F(a,b) ∧ F(b,c)) ⇒ F(a,c).
}
isEqRelation(P) ∧ isEqRelation(Q).
```

In existing treatments of templates, their semantics is given in a transformational way, essentially by translating them away as if they were macros. This appraoch has its limitations. An intellectually more gratifying view, certainly in a declarative setting, is that templates are higher order definitions. This allows for a much more general treatment. In some interesting cases, these higher order template definitions are recursive (see Example 4). In others, like the template symbol $tc(P, Q)$ specifying P as the transitive closure of Q, the definiens is itself an inductive definition (see Example 2) nested in the template definition of tc.

The goal of this work is to introduce a declarative template mechanism for the language FO(·). This logic posesses an expressive first order definition construct in the form of rules under well-founded semantics which was shown suitable to express informal definitions of the most common types (Denecker and Vennekens 2014). We want to extend FO(·)'s definition construc of to nested higher order definitions.

In the first part of this paper, we present a compositional framework for building an infinite class of logics. This framework specifies a principled way for building rule formalisms under well-founded and stable semantics from arbitrary logics, and ways to compose and nest arbitrary language constructs including higher order symbols, rule sets and aggregates. In the second part we use this framework to build a template formalism. As a last contribution we show that under suitable conditions, the standard approach of templates as rewriting macros also works in this formalism, thus recovering the results of existing approaches.

2 Related Work

Abstraction techniques have been an important area of research since the dawn of programming (Shaw 1984). Popular programming languages such as C++ consider templates as a keystone for abstractions (Musser et al. 2009). Within the ASP community, work by Ianni et al. (Ianni et al. 2004) and Baral et al. (Baral et al. 2006) introduced concepts to support composability, called templates and macros respectively. The key idea is to abstract away common constructs through the definition of generic 'template' predicates. These templates can then be resolved using a rewriting algorithm.

More formal attempts at introducing more abstractions in ASP were made. Dao-Tran et al. introduced modules which can be used in similar ways as templates (Dao-Tran et al. 2009) but has the disadvantage that his template system introduces additional computational complexity, so the user has to be very careful when trying to write an efficient specification.

Previously, meta-programming (Abramson and Rogers 1989) has also been used to introduce abstractions, for example in systems such as HiLog (Chen et al. 1993). One of HiLogs most notable features is that it combines a higher-order syntax with a first-order semantics. HiLogs main motivation for this is to introduce a useful degree of second order yet remain decidable. While decidability is undeniably an interesting property, the problem of decidability already arises in logic programs under well-founded or stable semantics, certainly with the inclusion of inductive definitions: the issue of undecidability is not inherent to the addition of template behavior. As a result, in recent times deduction inference has been replaced by various other, more practical inference methods such as model checking, model expansion, or querying. Furthermore, for practical applications, we impose the restriction of stratified templates for which an equivalent first-order semantics exists.

An alternative approach is to see a template instance as a call to another theory, using another solver as an oracle. An implementation of this approach exists in HEX (Eiter et al. 2011). This implementation however suffers from the fact that the different calls occur in different processes. As a consequence, not enough information is shared which hurts the search. This is analog to the approach presented in (Tasharrofi and Ternovska 2011), where a general approach to modules is presented. A template would be an instance of a module in this framework, however the associated algebra lacks the possibility to quantify over modules.

Previous efforts where made to generalize common language concepts, such as the work by Lifschitz (Lifschitz 1999) who extended logic programs to allow arbitrary nesting of conjunction $\wedge$, disjunction $\vee$ and negation as failure in rule bodies. The nesting in this paper is of very different kind, by allowing the full logic, including definitions itself, in the body.

3 Preliminaries

Symbols. We assume an infinite supply of (typed) symbols. A vocabulary Σ is a set of (typed) symbols. For each symbol σ, $\tau(\sigma)$ is its type. For a tuple $\bar{\sigma}$, $\tau(\bar{\sigma})$ denotes the tuple of types.

An untyped logic is one with a single type. But for the purposes of this paper, it is natural to use at least a simple form of typing, namely to distinguish between first order symbols and the second order (template) symbols. We distinguish between base types (some of which may be interpreted, e.g., $\mathbb{B}, \mathbb{Z}$) and composite types. A simple type system that suffices for this paper consists of the following types:

- base types δ and $\mathbb{B}$; δ represents the domain;
- first order types: n-ary predicate types $\delta^n \rightarrow \mathbb{B}$ and function types $\delta^n \rightarrow \delta$. As usual, propositional symbols and constants are predicate and function symbols of arity $n = 0$.
- second order types: n-ary predicate types $(\tau_1, \dots, \tau_n) \rightarrow \mathbb{B}$ with each τ_i a first order type or δ.

This is the type system that we have in mind in this paper. It suffices to handle untyped first order logic and second order predicates (no second order functions are needed). However, the framework below is well-defined for much richer type systems (including higher order types, type theory).

(Partial) values. For interpreted base types, there is a fixed domain of values. E.g., the domain of the boolean type $\mathbb{B}$ is $\mathcal{T}wo = \{\mathbf{t}, \mathbf{f}\}$. For other base types τ, the domain of values is chosen freely. For composite types, the set of values is constructed from the values of the base types.

For the simple type system above, the values of all types are determined by the choice of the domain associated with δ. For any domain D, we can define the domain τ^D of any type τ as follows:

- $\delta^D = D$, $\mathbb{B}^D = \mathcal{T}wo$
- first order predicates: $(\delta^n \to \mathbb{B})^D$ is the set of all functions from D^n to $\mathcal{T}wo$ (or equivalently, the set of all subsets of D^n). For first order functions, $(\delta^n \to \delta)^D$ is the set of all functions from D^n to D.
- second order predicates: $((\tau_1, \ldots, \tau_n) \to \mathbb{B})^D$ is the set of all functions from ${\tau_1}^D \times \cdots \times {\tau_n}^D$ to $\mathcal{T}wo$.

To define the semantics of inductive definitions, partial values for predicates are essential (since only predicates are defined in the logics of this paper, we do not introduce partial values for functions). A *partial set* on domain D is a function from D to $\mathcal{T}hree = \{\mathbf{t}, \mathbf{u}, \mathbf{f}\}$. A *partial* value of a predicate type $\tau' = (\bar{\tau} \to \mathbb{B})$ in domain D is a *partial set* with domain $\bar{\tau}^D$. $\mathcal{T}hree$ extends $\mathcal{T}wo$ and is equipped with two partial orders: the truth order $\leq$ is the least partial order satisfying $\mathbf{f} \leq \mathbf{u} \leq \mathbf{t}$, the precision order $\leq_p$ the least partial order satisfying $\mathbf{u} \leq_p \mathbf{f}, \mathbf{u} \leq_p \mathbf{t}$. The orders $\leq$ and $\leq_p$ on $\mathcal{T}hree$ are pointwise extended to partial sets. $\mathbf{u}$ is seen as an approximation of truth values, not as a truth value in its own right. A partial set that is maximally precise has range $\mathcal{T}wo$ and is called *exact.* A partial set $\mathcal{S}$ is seen as an approximation of any exact set S for which $\mathcal{S} \leq_p S$.

(Partial) Interpretations. A partial Σ-interpretation $\mathcal{I}$ consists of a suitable domain $\tau^{\mathcal{I}}$ for every type τ in Σ (which is the set of partial sets on ${\tau_d}^{\mathcal{I}}$ in case τ is a predicate type with domain type τ_d), and for every symbol $\sigma \in \Sigma$ of type τ a value $\sigma^{\mathcal{I}} \in \tau^{\mathcal{I}}$. An exact Σ-interpretation is one that assigns exact values. The class of partial Σ-interpretations is denoted $\mathcal{I}nt(\Sigma)$; the class of exact Σ-interpretations is $Int(\Sigma)$.

The precision order $\leq_p$ and truth order $\leq$ are extended to partial interpretations in the standard way: $\mathcal{I} \leq_p \mathcal{I}'$ if $\mathcal{I}, \mathcal{I}'$ interpret the same vocabulary Σ, have the same values for all types and non-predicate symbols, and $P^{\mathcal{I}} \leq_p P^{\mathcal{I}'}$ for every predicate symbol $P \in \Sigma$. Likewise for the truth order $\leq$. We use $\mathcal{I}$ to denote a partial interpretation (which may be exact) and I to denote an exact interpretation.

The restriction of a Σ-interpretation $\mathcal{I}$ to $\Sigma' \subseteq \Sigma$ is denoted as $\mathcal{I}|_{\Sigma'}$. If $\mathcal{I}$ is a partial Σ-interpretation, σ a symbol (that might not belong to Σ) and v a well-typed value for σ, then $\mathcal{I}[\sigma : v]$ is the $(\Sigma \cup \{\sigma\})$-interpretation identical to $\mathcal{I}$ except that v is the value of σ.

Given an interpretation $\mathcal{I}$ of at least the types of Σ, a *domain atom* of an n-ary predicate symbol $P \in \Sigma$ of type $\bar{\tau} \to \mathbb{B}$ in $\mathcal{I}$ is a pair $(P, \overline{d})$ where $\overline{d} \in \bar{\tau}^{\mathcal{I}}$.

It is denoted as $P(\overline{d})$. If $\mathcal{I}$ interprets P, a domain atom $P(\overline{d})$ has a truth value $P(\overline{d})^{\mathcal{I}} = P^{\mathcal{I}}(\overline{d})$.

For any $v \in \mathcal{T}hree$ and set X of domain atoms of partial interpretation $\mathcal{I}$, we denote $\mathcal{I}[X : \mathbf{t}]$ the interpretation identical to $\mathcal{I}$ except that each $A \in X$ is true; similarly for $\mathcal{I}[X : \mathbf{u}], \mathcal{I}[X : \mathbf{f}]$. We may concatenate such notions and write $\mathcal{I}[X : \mathbf{u}][Y : \mathbf{f}]$, with the obvious meaning (first revising X, next revising Y).

Logics $\mathcal{L}^{\upsilon}$. A logic is specified as a pair $(\mathcal{L}, \upsilon)$ (denoted $\mathcal{L}^{\upsilon}$) such that $\mathcal{L}$ is a function mapping vocabularies Σ to sets $\mathcal{L}(\Sigma)$ of expressions over Σ, and υ is a two-valued or three-valued truth assignment. An expression φ of $\mathcal{L}(\Sigma)$ has free symbols in Σ; it may contain other symbols provided they are bound by some scoping construct in a subexpression of φ (e.g., a quantifier). If $\Sigma \subseteq \Sigma'$, then $\mathcal{L}(\Sigma) \subseteq \mathcal{L}(\Sigma')$.

A (three-valued) truth assignment υ maps tuples $(\varphi, \mathcal{I})$ where $\mathcal{I}$ interprets all free symbols of φ, to $\mathcal{T}hree$. This function satisfies the following properties: (1) if $\varphi \in \mathcal{L}(\Sigma)$, $\Sigma \subseteq \Sigma'$ and $\mathcal{I}$ is a Σ'-interpretation, then $\varphi^{\upsilon:\mathcal{I}} = \varphi^{\upsilon:\mathcal{I}|_{\Sigma}}$; (2) exactness: $\varphi^{\upsilon:I} \in \mathcal{T}wo$ for every exact interpretation I; (3) $\leq_p$-monotonicity: if $\mathcal{I} \leq_p \mathcal{I}'$ then $\varphi^{\upsilon:\mathcal{I}} \leq_p \varphi^{\upsilon:\mathcal{I}'}$. A two-valued truth assignment υ is defined only for exact interpretations and satisfies (1) and (2).

Definition 3.1. We say that two formulas φ_1 and φ_2 over Σ_1 and Σ_2 respectively are Σ-*equivalent*, with $\Sigma \subseteq (\Sigma_1 \cap \Sigma_2)$, if for any interpretation I over Σ, there exists an expansion I_1 to Σ_1 for which $\varphi^{\upsilon:I_1} = \mathbf{t}$ iff there exists an expansion I_2 of I to Σ_2 for which $\varphi^{\upsilon:I_2} = \mathbf{t}$. If in addition $\Sigma_1 = \Sigma_2 = \Sigma$, we call φ_1 and φ_2 equivalent; hey have the same truth value in all Σ-interpretations.

4 Well-founded and stable semantics for $\mathcal{L}$-rule sets

In this section, we show that for each logic $\mathcal{L}^{\upsilon}$ with a three-valued truth assignment υ, it is possible to define a rule logic under a well-founded and under a stable semantics. Let $\mathcal{L}^{\upsilon}$ be such a logic.

Definition 4.1. An $\mathcal{L}$-rule over Σ is an expression $\forall \overline{x}(P(\overline{x}) \leftarrow \varphi)$ with P a predicate symbol in Σ, $\overline{x}$ a tuple of "variable" symbols and $\varphi \in \mathcal{L}(\Sigma \cup \{\overline{x}\})$. An $\mathcal{L}$-rule set over Σ is a set of $\mathcal{L}$-rules over Σ. Rule sets will be denoted with Δ.

The set *Def*(Δ) is the set of predicate symbols $P \in \Sigma$ that occur in the head of a rule. *Par*(Δ) is the set of all other symbols that occur in Δ. Elements of *Def*(Δ) are called *defined* symbols, the other ones are called *parameters* of Δ.

Definition 4.2. A context $\mathcal{O}$ of a $\mathcal{L}$-rule set Δ is a $\Sigma \setminus Def(\Delta)$-interpretation.

For a given context $\mathcal{O}$, the set $\{\mathcal{I} \mid \mathcal{I}|_{Par(\Delta)} = \mathcal{O}\}$ of partial Σ-interpretations expanding $\mathcal{O}$ is isomorphic to the set of partial sets of domain atoms of *Def*(Δ) in $\mathcal{O}$. Thus, given $\mathcal{O}$, a partial set of domain atoms specifies a unique partial interpretation $\mathcal{I}$ expanding $\mathcal{O}$ and vice versa.

We call a set of domain atoms a $\mathbf{t}$-set, respectively $\mathbf{u}$-set, $\mathbf{f}$-set of partial interpretation $\mathcal{I}$ if its elements have truth value $\mathbf{t}$, respectively $\mathbf{u}$, $\mathbf{f}$ in $\mathcal{I}$.

Definition 4.3. A partial interpretation $\mathcal{I}$ is closed under Δ if for any domain atom $P(\overline{d})$ and rule $\forall\overline{x}(P(\overline{x}) \leftarrow \varphi) \in \Delta$, if $\varphi[\overline{d}]^{\upsilon:\mathcal{I}} = \mathbf{t}$ then $P(\overline{d})^{\upsilon:\mathcal{I}} = \mathbf{t}$.

Definition 4.4. An unfounded set of Δ in $\mathcal{I}$ is a $\mathbf{u}$-set U of defined domain atoms in $\mathcal{I}$, for which every atom $P(\overline{d}) \in U$ and rule $\forall\overline{x}(P(\overline{x}) \leftarrow \varphi) \in \Delta$, $\varphi[\overline{d}]^{\upsilon:\mathcal{I}[U:\mathbf{f}]} = \mathbf{f}$.

Definition 4.5. A partial interpretation $\mathcal{I}$ extending context $\mathcal{O}$ is a *partial stable interpretation* of Δ if

1. for each domain atom $P(\overline{d})$, $P(\overline{d})^{\mathcal{I}} = Max_{\leq}\{\varphi[\overline{d}]^{\upsilon:\mathcal{I}} \mid \forall\overline{x}(P(\overline{x}) \leftarrow \varphi) \in \Delta\}$;
2. (prudence) there exists no non-empty $\mathbf{t}$-set T and no (possibly empty) $\mathbf{u}$-set U of $\mathcal{I}$ such that $\mathcal{I}[T : \mathbf{u}][U : \mathbf{t}]$ is closed under Δ;
3. (braveness) the only unfounded set of Δ in $\mathcal{I}$ is $\emptyset$.

Definition 4.6. We call a partial interpretation $\mathcal{I}$ a well-founded interpretation of Δ if $\mathcal{I}$ is the $\leq_p$-least partial stable model $\mathcal{I}'$ of Δ such that $\mathcal{I}'|_{Par(\Delta)} = \mathcal{I}|_{Par(\Delta)}$.

Definition 4.7. We call an (exact) interpretation I a stable interpretation of Δ if I is an exact partial stable model of Δ.

Given that a stable I has only the empty $\mathbf{u}$-set, conditions (2) and (3) simplify to that there is no non-empty $\mathbf{t}$-set T of $\mathcal{I}$ such that $\mathcal{I}[T : \mathbf{u}]$ is closed under Δ.

Proposition 4.8. *Let φ, φ' be equivalent under υ (same truth value in all partial interpretations). Then substituting φ for φ' in the body of a rule of Δ preserves the class of partial stable (hence, well-founded and stable) interpretations.*

Proof. This is trivial, since the conditions of partial stable interpretation are defined in terms υ, which cannot distinguish φ from φ'. □

Two logics. Using the above two concepts we define two rule logics. Expressions in both logics are the same: finite sets of rules.

Definition 4.9. For logic $\mathcal{L}^{\upsilon}$, we define logic $R(\mathcal{L}^{\upsilon})^{w}$ where $R(\mathcal{L}^{\upsilon})(\Sigma)$ is the collection of finite rule sets over Σ and w the two-valued truth assignment defined as $\Delta^{w:I} = \mathbf{t}$ if I is an exact well-founded interpretation of Δ and $\Delta^{w:I} = \mathbf{f}$ otherwise.

Definition 4.10. For logic $\mathcal{L}^{\upsilon}$, we define the logic $R(\mathcal{L}^{\upsilon})^{st}$ where $R(\mathcal{L}^{\upsilon})(\Sigma)$ is as above and st is the two-valued truth assignment defined as $\Delta^{st:I} = \mathbf{t}$ if I is an exact stable interpretation of Δ and $\Delta^{st:I} = \mathbf{f}$ otherwise.

For the logic FO^{k}, with FO first order logic and k the 3-valued Kleene truth assignment (Kleene 1952), the rule formalism $R(FO^{k})^{w}$ corresponds to the (formal) definitions in the logic FO(ID) (Denecker 2000; Denecker and Ternovska 2008) while the formalism $R(FO^{k})^{st}$ corresponds to the rule formalism in the logic ASP-FO (Denecker et al. 2012).

In (Denecker and Vennekens 2014), the relation between the main forms of (informal) *definitions* found in mathematical text, and rule sets in $R(FO^{k})^{w}$ was analyzed. Not all rule sets of $R(FO)$ express sensible (informal) definitions, but for those that do, the well-founded interpretations are exact and correctly specify the defined sets. Therefore, a rule set Δ was called a *paradox-free* or *total definition* in an exact context $\mathcal{O}$ if its well-founded interpretation expanding $\mathcal{O}$ is exact. For paradox-free rule sets, w and st coincide. Important classes of rule sets are always paradox-free: non-recursive, monotone inductive rule sets, and rule sets by ordered or iterated induction over some well-founded induction order as defined in (Denecker and Vennekens 2014).

5 Compositional framework for building logics with definitions

This section effectively defines an infinite collection of logics. We define compositional constructs which add new expressions, such as definitions, to an existing logic. By iterating such extension steps, these constructs can be nested.

5.1 Approximating boolean functions

We frequently need to extend a boolean function defined on a domain X of exact values (e.g., $\mathcal{T}wo$, exact sets, exact interpretations, or tuples including these) to the domain $\mathcal{X}$, $\leq_p$ of partial values. Examples are the boolean functions associated with connectives $\neg, \wedge, \ldots$, or the truth assigments w and st of $R(\mathcal{L}^v)$ defined on $Int(\Sigma)$. Given such a function $F : X \to \mathcal{T}wo$, we search for an approximation $\mathcal{F} : \mathcal{X} \to \mathcal{T}hree$ such that:

- $\leq_p$-montone: if $\mathbf{x} \leq_p \mathbf{y} \in \mathcal{X}$ then $\mathcal{F}(\mathbf{x}) \leq_p \mathcal{F}(\mathbf{y})$;
- exact and extending: for $x \in X$, $\mathcal{F}(x) = F(x)$.

Definition 5.1. We define the ultimate approximation $\widetilde{F} : \mathcal{X} \to \mathcal{T}hree$ of F by defining $\widetilde{F}(\mathbf{x}) = \text{glb}_{\leq_p} \{F(x) \mid \mathbf{x} \leq_p x\} \in \mathcal{T}hree$.

Proposition 5.2. *$\widetilde{F}$ is the most precise $\leq_p$-monotone exact extension of F.*

Proof. $\leq_p$-monotonicity follows from the transitivity of $\leq_p$. Exactness, from the fact that elements of $Int(\Sigma)$ are maximally precise. That $\widetilde{F}$ is the most precise approximation of F is clear as well. □

Several important examples follow. For a standard connective $c \in \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$ with corresponding boolean function $\mathbf{c} : \mathcal{T}wo^n \to \mathcal{T}wo$, the function $\widetilde{\mathbf{c}} : \mathcal{T}hree^n \to \mathcal{T}hree$ is the three-valued truth function used in the Kleene truth assignment k.

The semantics of quantifiers $\forall, \exists$ and generalized quantifiers such as aggregates are given by functions on sets (or tuples including sets). E.g., for quantification over domain D these are the boolean functions $\forall_D, \exists_D$ defined $\forall_D(S) = \mathbf{t}$ iff $D \subseteq S$, and $\exists_D(S) = \mathbf{t}$ iff $D \cap S \neq \emptyset$. Two commonly used numerical aggregate functions are cardinality $\#$ and *sum* (the latter mapping (finite) sets S of tuples $\bar{d}$ to $\sum_{\bar{d} \in S} d_1$). For every numerical aggregate function Agg and boolean operator $\sim \in \{=, <, >\}$ on numbers, the boolean function $Agg_\sim$ maps tuples (S, n) to $\mathbf{t}$ iff $Agg(S){\sim}n$.

For all these higher order boolean functions F, $\widetilde{F}$ is the most precise approximation on three-valued sets. The three-valued aggregate functions $\widetilde{Agg_\sim}$ were introduced originally in (Pelov et al. 2007) to define stable and well-founded semantics for aggregate logic programs. The functions $\widetilde{\forall}_D, \widetilde{\exists}_D$ are used in the Kleene truth assignment k: let D be $\tau(x)^{\mathcal{I}}$, and $\mathcal{S} = \{(d, \varphi^{k:\mathcal{I}[x:d]}) \mid d \in D\}$, i.e. the three valued set mapping domain elements $d \in D$ to $\varphi^{k:\mathcal{I}[x:d]})$, one defines $(\forall x\ \varphi)^{k:\mathcal{I}} = \widetilde{\forall}_D(\mathcal{S}) = Min_{\leq}\{\mathcal{S}(d) \mid d \in D\} = Min_{\leq}\{\varphi^{k:\mathcal{I}[x:d]}) \mid d \in D\}$.

For any two-valued truth assigment v on $\mathcal{L}$, $\widetilde{v}$ is a sound three-valued truth assignment. In case of FO and its truth assignment v, $\widetilde{v}$ was introduced in (van Fraassen 1966) where it was called the supervaluation s. s is not truth functional, for if $p^{\mathcal{I}} = q^{\mathcal{I}} = \mathbf{u}$, then $(p \vee \neg p)^{s:\mathcal{I}} = \mathbf{t} \neq (p \vee q)^{s:\mathcal{I}} = \mathbf{u}$ while the components of the two disjunctions have the same supervaluation. A truth-functional definition of a three-valued truth assignment is obtained by using the ultimate approximations of the boolean functions associated to connectives and quantifiers. This yields exactly the Kleene truth assigment k. It is $\leq_p$-monotone, exact and extending, and strictly less precise than s as can be seen from $(p \vee \neg p)^{k:\mathcal{I}} = \mathbf{u}$: the supervaluation "sees" the logical connection between p and $\neg p$ in this tautology while k does not.

Other applications serve to extend, for arbitrary logic $\mathcal{L}^v$, the two-valued well-founded and stable truth assignments w and st on $R(\mathcal{L}^v)$ to three-valued extensions $\widetilde{w}, \widetilde{st}$. Here, it holds that $\Delta^{\widetilde{w}:\mathcal{I}} = \mathbf{t}$ (respectively $\mathbf{f}$) if every (respectively, no) instance I of $\mathcal{I}$ is a well-founded interpretation of Δ.

5.2 Composing logics by combining logic constucts

A standard way of defining the syntax of a logic is through a set of often inductive syntactical rules, typically described in Backus Naur Form (BNF). The truth assignment v is then defined by recursion over the structure of the expressions. Below, we identify a language construct C with a pair of a syntactical and a semantical rule. The rules below construct, for a language construct C, a new logic $C(\mathcal{L}^v)^{v'}$ with expressions obtained by applying C on subexpressions of $\mathcal{L}^v$ and v' a truth assignment for $C(\mathcal{L})$. Afterwards, complex logics with multiple and nested language constructs can be built by iterating these construction steps.

- Atom^1 and Atom^2: for first order predicates p and second order ones P respectively. $\bar{t}$ is a tuple of terms, $\overline{x}$ of variables.

$$Atom^1 ::= p(\bar{t}) \text{ where } p(\bar{t})^{v':\mathcal{I}} = p^{\mathcal{I}}(t_1^{\mathcal{I}}, \cdots, t_n^{\mathcal{I}})$$

$$Atom^2 ::= P(\overline{x}) \text{ where } P(\overline{x})^{v':\mathcal{I}} = P^{\mathcal{I}}(x_1^{\mathcal{I}}, \cdots, x_n^{\mathcal{I}})$$

- N-ary connectives $c \in \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$,

$$c(\mathcal{L}^v) ::= c(\alpha_1, \ldots, \alpha_n) \text{ where } c(\alpha_1, \ldots, \alpha_n)^{v':\mathcal{I}} = \widetilde{\mathbf{c}}({\alpha_1}^{v:\mathcal{I}}, \cdots, {\alpha_n}^{v:\mathcal{I}})$$

- Generalized quantifiers $C \in \{\forall, \exists, Agg_{\sim}\}$. Below, $C\langle\overline{x}, \alpha, z\rangle$ denotes the syntactic expression, e.g., $\forall\langle x, \alpha\rangle$ is $\forall x\ \alpha$; $Agg_{\sim}\langle\overline{x}, \alpha, z\rangle$ is $Agg\{\overline{x} : \alpha\}{\sim}z$.

$$C(\mathcal{L}^v) ::= C\langle\overline{x}, \alpha, z\rangle \text{ where } (C\langle\overline{x}, \alpha, z\rangle)^{v':\mathcal{I}} = \widetilde{C}(\{(\overline{d}, \alpha^{v:\mathcal{I}[\overline{x}:\overline{d}]}) | \overline{d} \in \tau(\overline{x})^{\mathcal{I}}\}, z^{\mathcal{I}})$$

- Definitions as rule sets (R_w) (similarly, one could define R_{st}):

$$R_w(\mathcal{L}^v) ::= R(\mathcal{L}) \text{ where } \Delta^{v':\mathcal{I}} = \Delta^{\widetilde{w}:\mathcal{I}}$$

where w is the well-founded assignment of $R(\mathcal{L}^v)$ as defined in Section 4.

Building logics. Using the above rules of language constructs, an (infinite) class of logics with three-valued semantics can be built. Moreover, every combination of the above rules gives rise to a valid three-valued truth assignment.

Proposition 5.3. *Every (sub)set of the above language constructs (possibly closed under recursive application) defines a logic with a proper three-valued truth assignment (i.e. it is $\leq_p$-monotone, exact and extending).*

For example, given a logic $\mathcal{L}^v$, we define $\mathcal{L}'^{v'} = R(\mathcal{L}^v)^{\widetilde{w}}$ by one application of R_w on $\mathcal{L}^v$. By iterating R_w, logics $(R(\mathcal{L}^v)^{\widetilde{w}})^n$ with nested definitions are built. Every BNF in terms of the above language constructs now implicitly defines a three-valued logic. The definition of first order logic FO^k with k the standard three-valued Kleene truth assignment, can be descibed in BNF or more compactly as $\{Atom^1, \wedge, \vee, \neg, \forall, \exists\}^*$ (here $*$ indicates recursive application of the construction rules). The logic FO(ID) defined in (Denecker and Ternovska 2008) is the union of logics FO^k and $R(FO^k)^{\widetilde{w}}$. A further extension is the new logic $FO(ID^*) = \{Atom^1, \wedge, \vee, \neg, \forall, \exists, R_w\}^*$ which has definitions nested in formulas and definition rule bodies. A logic in which templates can easily be embedded is $SO(ID^*) = \{Atom^1, Atom^2, \wedge, \vee, \neg, \forall, \exists, R_w\}^*$. It is a second order extension of FO(ID) which allows for nesting of definitions in rule bodies.

6 Templates

We envision a library of application independent templates in the form of second order definitions that encapsulate prevalent patterns and concepts and that can be used as building blocks to compose logic specifications. Below, we formally define the concepts and show that non-recursive templates do not increase the computational complexity of FO(ID^*) and can be eliminated by a rewriting process.

6.1 Definition and usage

We assume the existence of a set of template symbols. A template is a context-agnostic second order definition of template symbols. As such it should define and contain only domain independent symbols: interpreted symbols and template symbols. A template might define a template symbol in terms of other template symbols and interpreted predicates, but not in terms of user-defined symbols.

Definition 6.1. The template vocabulary Σ_{Temp} is the vocabulary consisting of all interpreted symbols (such as arithmetic symbols) and all template symbols.

Definition 6.2. A template is a second order definition Δ over Σ_{Temp} such that *Def*(Δ) consists of template symbols.

Thus the set of parameters *Par*(Δ) of a template consists only of interpreted symbols and template symbols.

The concepts used in Example 1 are now fully defined. Another common example is the template tc expressing that Q/2 is the transitive closure of P/2, as shown in Example 2. Note that this example cannot be written without a definition in the body of the template, so this further motivates our choice to allow definitions in the bodies of other definitions in our recursive construction of the logic SO(ID^*).

Example 2: This template TC expresses that Q is the transitive closure of P

```
{tc(P,Q) ←
  {Q(x,y) ← P(x,y) ∨(∃ z: Q(x,z)∧Q(z,y))}.
}
```

Another notable aspect of this approach to templates is that recursive templates are well-defined. This enables us to write recursive templates, for example to define a range:

Example 3: P is the range of integers from a to b

```
{range(P, a, b) ←
  {P(a).
   P(x) ← a < b ∧ (∃ Q : range(Q,a+1,b) ∧ Q(x)).
  }
}
```

It is possible to rewrite Example 3 into a non-recursive template. Example 4 contains an example which is not rewritable in such a way.

Example 4: cur is a winning position in a two-player game

```
{win(cur,Move, IsWon) ← IsWon(cur) ∨
  ∃ nxt : Move(cur,nxt) ∧ lose(nxt,Move,IsWon).
 lose(cur,Move, IsWon) ← ¬IsWon(cur) ∧
  ∀ nxt : Move(cur,nxt) ⇒ win(nxt,Move,IsWon).
}
```

This template defining *win* and *lose* by simultaneous definition, is a monotone second order definition and has a two-valued well-founded model. That it cannot be rewritten without recursion over second order predicates follows from the fact that deciding if a tuple belongs to a non-recursively defined second order predicate is in PH while deciding winning positions in generalized games is harder (if the polynomial hierarchy does not collapse) and this last problem corresponds to deciding elementship in the relation win defined in Example 4.

Definition 6.3. A template library L is a finite set of templates satisfying (1) every template is paradox-free; (2) every template symbol is defined in exactly one template; (3) the set of templates is hierarchically stratified: there is a strict order $<$ on template symbols such that for each $\Delta \in L$, if $P \in \mathit{Def}(\Delta), Q \in \mathit{Par}(\Delta)$ then $Q < P$.

Proposition 6.4. *For a template library L, each interpretation I not interpreting symbols of Σ_{Temp} has a unique two-valued expansion I' to Σ_{Temp} that satisfies L.*

Proof. By induction on the hierarchy $<$ of L. □

6.2 The Σ_{Temp} vocabulary restriction

The condition that templates should be built from Σ_{Temp} and not from user-defined symbols is to ensure that templates are domain independent 'drop-in' building blocks. This restriction might seem too stringent, but we can show that many template definitions for which it does not hold, can be rewritten as an equivalent one for which it holds.

Let Δ be definition of second order predicates with $Def(\Delta) \cap \Sigma_{Temp} = \emptyset$, and $\overline{o}$ the tuple of all free (user-defined) symbols of $Par(\Delta) \setminus \Sigma_{Temp}$ (arranged in some arbitrary order). For such definitions, we define a templified version. For any rule or formula Ψ, we define $\Psi^{\overline{o}}$ to be Ψ except that every atom $P(\overline{t})$ in Ψ with $P \in Def(\Delta)$ is replaced by $P'(\overline{t}, \overline{o})$, with P' a new symbol extending P with new arguments corresponding to $\overline{o}$.

We say that a structure I corresponds to I' if I, I' interpret the free symbols of Δ, respectively those of Δ_{Temp}, they are identical on shared symbols and for each $P \in Def(\Delta)$, $P^I = \left\{\overline{d} | (\overline{d}, \overline{o}^I) \in P'^{I'}\right\}$. Note that for each I' and each value $\overline{d}_o$ for $\overline{o}$ in the domain of I', there is a unique interpretation I with $\overline{o}^I = \overline{d}_o$ that corresponds to I'.

Definition 6.5. We define the *templified definition* Δ_{Temp} of Δ as the definition $\{\forall \overline{o}(\Psi^{\overline{o}}) \mid \Psi \in \Delta\}$ and we define $\Sigma'_{Temp} = \Sigma_{Temp} \cup \{P' \mid P \in Def(\Delta)\}$.

We assume that $\overline{o}$ consists only of first order predicate symbols. Under this condition, the templified definition Δ_{Temp} is a template over Σ'_{Temp}.

Proposition 6.6. *Let I be a well-founded model of Δ and I' a well-founded model of Δ_{Temp} such that I and I' are identical on Σ_{Temp}. Then it holds that*

$$P^I = \left\{\overline{d} | (\overline{d}, \overline{o}^I) \in P'^{I'}\right\}$$

Stated differently, I corresponds to I'. The templified definition captures the original one, and hence, each theory can be rewritten in terms of the new templified defined symbols.

Proof. Assume that I corresponds to I'. It is easy to prove, by induction on the formula structure, that for any formula φ in the vocabulary of Δ, it holds that $\varphi^I = (\varphi^{\overline{o}})^{I'[\overline{o}:\overline{o}^I]}$. We call this the independency property since it shows that $(\varphi^{\overline{o}})^{I'[\overline{o}:\overline{o}^I]}$ is influenced by only a small part of the interpretation of P', namely the values of domain atoms $P'(\overline{d}, \overline{o}^I)$.

The key property to prove is that I' is a partial stable interpretation of Δ_{Temp} iff for each value $\overline{d}_o$ for $\overline{o}$, the unique I that corresponds to I' such that $\overline{o}^I = \overline{d}_o$ is a partial stable interpretation of Δ. Intuitively, a partial stable interpretation of Δ_{Temp} is a kind of union of partial stable interpretations of Δ, one for each assignment of values to $\overline{o}$.

We prove this property only in one direction. The other direction is similar.

Assume that I' is a partial stable interpretation of Δ_{Temp} satisfying the three conditions of Definition 4.5. We need to show for every I that corresponds to I', that I is a partial stable interpretation of Δ. Condition 1), that $P(\overline{d})^I = Max_{\leq}\{\varphi[\overline{d}]^I \mid \forall\overline{x}(P(\overline{x}) \leftarrow \varphi[\overline{x}]) \in \Delta\}$ follows from the fact that $P'(\overline{d},\overline{o}^I)^{I'}$ satisfies the corresponding equation for Δ_{Temp}, that $P(\overline{d})^I = P'(\overline{d},\overline{o}^I)^{I'}$, and that for each rule body φ for P, $\varphi[\overline{d}]^I = (\varphi^{\overline{o}})^{I'[\overline{o}:\overline{o}^I]}$ (by the independency property). The condition 2) follows from the fact that when $T' = \{P'(\overline{d},\overline{o}^I) \mid P(\overline{d}) \in T\}$, and $U' = \{P'(\overline{d},\overline{o}^I) \mid P(\overline{d}) \in U\}$, then T is a **t**-set and U a **u**-set of I such that $I[T : \mathbf{u}][U : \mathbf{t}]$ is closed under Δ iff T' is a **t**-set and U' a **u**-set of I' such that $I'[T' : \mathbf{u}][U' : \mathbf{t}]$ is closed under Δ_{Temp}. This follows from the independency property. Condition 3) is proven similarly.

It is easy to see that this property entails the proposition, since intuitively, it entails that a well-founded model I' of Δ_{Temp}, which is the $glb_{\leq_p}$ of all partial stable interpretations of Δ_{Temp} with the same context as I', contains for each value $\overline{d}_o$ for $\overline{o}$ the $glb_{\leq_p}$ of the partial stable interpretations I of Δ in the context with $\overline{o}^I = \overline{d}_o$. □

6.3 Simple Templates

Extending a logic with arbitrary (recursive) templates may easily increase the descriptive complexity of the logic. Below, we develop a simple but useful template formalism for FO(ID) that does not have this effect. In addition, we show that libraries of simple templates can be compiled away using them as macros.

In Figure 1 we define sublanguages FO(ID^*), ESO(ID^*) and ASO(ID^*) of SO(ID^*) (by mutual recursion) consisting of atoms, negations, conjunctions, quantification, definitions and the let-construct. This last construct represents a second order quantification, where the quantified symbol(s) S are defined in an accompanying paradox-free definition Δ. Definitions of second order symbols in ESO(ID^*) and ASO(ID^*) contain only (possibly nested) first order definitions. Since model checking of (nested) first order definitions is polynomial, the descriptive complexity of FO(ID^*) is P, of ESO(ID^*) is NP and of ASO(ID^*) is co-NP.

FO(ID^*) φ ::=	ESO(ID^*) ϵ ::=	ASO(ID^*) α ::=
$\mid s(\overline{t})(\in \mathit{Atom}^1)$	$\mid S(\overline{t})(\in \mathit{Atom}^2)$	$\mid S(\overline{t})(\in \mathit{Atom}^2)$
$\mid \neg\varphi$	$\mid \neg\alpha$	$\mid \neg\epsilon$
$\mid \varphi \wedge \varphi$	$\mid \epsilon \wedge \epsilon$	$\mid \alpha \wedge \alpha$
$\mid \exists_{FO}\ s : \varphi$	$\mid \exists_{FO}\ s : \epsilon$	$\mid \exists_{FO}\ s : \alpha$
$\mid$let $\{s(\overline{t}) \leftarrow \varphi\}$ in φ	$\mid$let $\{s(\overline{t}) \leftarrow \varphi\}$ in ϵ	$\mid$let $\{s(\overline{t}) \leftarrow \varphi\}$ in α
$\mid \{s(\overline{t}) \leftarrow \varphi\}$	$\mid \{s(\overline{t}) \leftarrow \varphi\}$	$\mid \{s(\overline{t}) \leftarrow \varphi\}$
	$\mid \exists_{SO}\ s : \epsilon$	$\mid \forall_{SO}\ s : \alpha$

(a) FO(ID^*) (b) SO(ID^*) (c) ASO(ID^*)

Figure 1: The FO(ID^*), ESO(ID^*) and ASO(ID^*) subformalisms of SO(ID^*)

Definition 6.7. A *simple template* is a template of the form $\{\forall \overline{x}(P(\overline{x}) \leftarrow \varphi_P[\overline{x}])\}$ with $P(\overline{x}) \in Atom^2$ and $\varphi_P \in \mathrm{FO}(ID^*)$.

A simple template defines one symbol and contains one rule with an $\mathrm{FO}(ID^*)$ body. Let L be a template library over Σ_{Temp} consisting of non-recursive simple templates. Such a library is equivalent to the conjunction the completion of its definitions $\forall \overline{x}(P(\overline{x}) \Leftrightarrow \varphi_P[\overline{x}])$. We want to show that while using such libraries increases convenience, reuse, modularity, it does not increase complexity nor expressivity. Also, such libraries can be used in the common way, as macros.

Theorem 6.8. *For $\Sigma \cap \Sigma_{Temp} = \emptyset$, let φ be a $\mathrm{ESO}(ID^*)$ formula over $\Sigma \cup \Sigma_{Temp}$ that does not contain definitions of template symbols. There exists a polynomially larger $\mathrm{ESO}(ID^*)$ formula φ_1 over Σ that is Σ-equivalent to $\{\varphi\} \cup L$. There exists a polynomially larger $\mathrm{FO}(ID^*)$ formula φ_2 over an extension Σ_1 of Σ that is Σ-equivalent to $\{\varphi\} \cup L$.*

Proof. The formula φ_1 is obtained by treating L as a set of macros. We iteratively substitute template atoms $P(\overline{t})$ in φ by $\varphi_P[\overline{t}]$. This process is equivalence preserving. It terminates due to the stratification condition on L, and the limit is a polynomially larger $\mathrm{ESO}(ID^*)$ formula φ_1 in the size of φ (exponential in $\#(L)$) that is Σ-equivalent to $\{\varphi\} \cup L$.

To obtain φ_2, we apply the well-known transformation of moving existential quantifiers to the front and skolemising them. Second order quantifiers can be switched with first order ones using:

$$\forall_{FO}\, x : \exists_{SO} P : \varphi \Leftrightarrow \exists_{SO} P' : \forall_{FO}\, x : \varphi[P(\overline{t}) \backslash P'(\overline{t}, x)]$$

This process preserves Σ-equivalence. As only a polynomial number of steps are needed to transform the formula into this desired state, the size of the resulting formula is polynomially larger. □

Previous results in (Ianni et al. 2004) indicated that the introduction of simple, stratified templates does not introduce a significant performance hit. The above theorem recovers these efficiency results.

7 Conclusion

In this paper we developed a new way to define language constructs for a logic. New language constructs must combine a syntactical rule with a three-valued semantic evaluation. This three-valued semantic evaluation is subject to certain restrictions. Language constructs can then be arbitrarily combined to compose a logic. In particular, we construct $\mathrm{SO}(ID^*)$: a second order language with inductive definitions.

Using this language, it is easy to define templates as second order definitions. We conclude our paper with a rewriting scheme to show that, given some restrictions, templates do not increase the descriptive complexity of the host language.

In the future, we want to generalize our way of defining language constructs to allow functions and provide a more comprehensive type system. On the more practical side, we intend to bring our ideas into practice by extending the IDP(IDP 2013) system with simple templates.

References

Abramson, H. and Rogers, H. 1989. *Meta-programming in Logic Programming.* MIT Press.

Baral, C., Dzifcak, J., and Takahashi, H. 2006. Macros, macro calls and use of ensembles in modular answer set programming. In *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, 376–390.

Chen, W., Kifer, M., and Warren, D. S. 1993. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming 15,* 3, 187–230.

Dao-Tran, M., Eiter, T., Fink, M., and Krennwallner, T. 2009. Modular nonmonotonic logic programming revisited. In *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, 145–159.

Denecker, M. 2000. Extending classical logic with inductive definitions. In *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, Eds. LNCS, vol. 1861. Springer, 703–717.

Denecker, M., Lierler, Y., Truszczyński, M., and Vennekens, J. 2012. A Tarskian informal semantics for answer set programming. In *International Conference on Logic Programming (Technical Communications)*, A. Dovier and V. S. Costa, Eds. LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 277–289.

Denecker, M. and Ternovska, E. 2008. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log. 9,* 2 (Apr.), 14:1–14:52.

Denecker, M. and Vennekens, J. 2014. The well-founded semantics is the principle of inductive definition, revisited. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, C. Baral, G. De Giacomo, and T. Eiter, Eds. AAAI Press, 22–31.

Eiter, T., Krennwallner, T., and Redl, C. 2011. Hex-programs with nested program calls. In *Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers*, H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds. Lecture Notes in Computer Science, vol. 7773. Springer, 269–278.

Ianni, G., Ielpa, G., Pietramala, A., Santoro, M. C., and Calimeri, F. 2004. Enhancing answer set programming with templates. In *10th International Workshop on Non-Monotonic Reasoning (NMR 2004), Whistler, Canada, June 6-8, 2004, Proceedings*, J. P. Delgrande and T. Schaub, Eds. 233–239.

IDP 2013. The IDP system. `http://dtai.cs.kuleuven.be/krr/software`.

Kleene, S. C. 1952. *Introduction to Metamathematics.* Van Nostrand.

Lifschitz, V. 1999. Answer set planning. In *16th International Conference on Logic Programming, ICLP 1999, Las Cruces, New Mexico, USA, November 29 - December 4, 1999, Proceedings*, D. De Schreye, Ed. MIT Press, 23–37.

Musser, D. R., Derge, G. J., and Saini, A. 2009. *STL tutorial and reference guide: C++ programming with the standard template library.* Addison-Wesley Professional.

Pelov, N., Denecker, M., and Bruynooghe, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *Theory Pract. Log. Program. 7,* 3, 301–353.

Shaw, M. 1984. Abstraction techniques in modern programming languages. *IEEE Software 1,* 4, 10–26.

Tasharrofi, S. and Ternovska, E. 2011. A semantic account for modularity in multi-language modelling of search problems. In *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, C. Tinelli and V. Sofronie-Stokkermans, Eds. Lecture Notes in Computer Science, vol. 6989. Springer, 259–274.

van Fraassen, B. 1966. Singular terms, truth-value gaps and free logic. *Journal of Philosophy 63,* 17, 481–495.