Lambda Expressions in CASL Architectural Specifications

Mihai Codescu
Mihai.Codescu@dfki.de

DFKI GmbH Bremen

Abstract. CASL architectural specifications provide a way to specify the structure of the implementations of software systems. Their semantics has been introduced in two manners: the first is purely model-theoretic and the second attempts to discharge model semantics conditions statically based on a diagram of dependencies between components (extended static semantics). In the case of lambda expressions, which are used to define the way generic units are built, the two semantics do not agree. We present a number of situations of practical importance when the current situation is unsatisfactory and propose a series of changes to the extended static semantics to remedy this.

1 Introduction

An idealized view on the process of software development would be to start with a requirement specification (most likely structured) and then to proceed with an architectural design describing the expected structure of the implementation (which can be different from the one of the specification). Architectural specifications in CASL [3] have been introduced as means of providing structure for the implementation: each architectural specification contains a number of components together with a linking procedure which describes how to combine the components to obtain an implementation of the overall system. (In contrast, the models of a structured specification are monolithic and have no more structure than models of basic specifications).

In the figure on the right, SP is the initial specification, $U_1, \dots U_n$ are the components of the architectural specifications with their specifications $SP_1, \dots, SP_n$ and k is the linking procedure involving the units, while the refinement relation is denoted $\rightsquigarrow$. The specification of each component can then play the role of requirement specification and the entire process repeats until specifications that can be easily translated into a program are reached. The only interaction allowed between components is the one contained in the architectural specification they are part of, that acts as an interface for them; this allows for a separation of implementation tasks, which can be performed independently.

$$SP \rightsquigarrow k \begin{cases} U_1 : SP_1 \\ \vdots \\ U_n : SP_n \end{cases}$$

The semantics of architectural specifications relies on compatibility checks between units as prerequisite for combining them. The intuitive idea is that

shared symbols must be interpreted in the same way for two models to be put together. The rules have been presented in two ways: the first is to define a basic static semantics and model semantics in a purely model-theoretical fashion and the compatibility checks are required in the model semantics whenever needed, while the second is an extended static semantics analysis which builds a graph of dependencies between units and discards the compatibility conditions statically. We briefly recall the two semantics and the relationships between them in Section 2. Units of an architectural specification can be *generic* [10], with the intended intuitive meaning that the implementation of the result specification depends on the implementations of the arguments (e.g. some auxiliary functions). Generic units are built using generic unit expressions, written in CASL using the λ-notation: $\lambda\ X_1 : SP_1,\ \ldots,\ X_n : SP_n\ .\ UT$, where UT is a unit term which contains $X_1, \ldots X_n$.

The motivation of this paper is rather technical: the extended static semantics rule for generic unit expression does not keep track of the dependencies between the units used in the unit term UT. This is unsatisfactory for a number of reasons that we give in detail in Section 3: first, the completeness theorem for extended static semantics (Theorem 5.4 in [4]) no longer holds when the language is extended with definitions of parametric units. Moreover, *unit imports* are known to be introducing complexity in semantics and verification of architectural specifications. One way to reduce complexity is to replace unit imports with an equivalent construction as below, provided that M is made visible locally in the anonymous architectural specification:

units M : SP1; N : SP2 **given** M; ...	is equivalent to	**units** M : SP1; N : **arch spec** { **units** F : SP1 → SP2 **result** F[M]}; ...

If N would be a generic unit, then the result of the architectural specification in the right side would be a λ-expression and the two constructions would no longer be equivalent because they treat differently the dependency between M and N. In Section 4 we present our proposed changes for the extended static semantics of architectural specifications, followed by a discussion in Section 4.1 on how the completeness result can be extended to cover lambda expressions as well. Section 4.2 further extends the changes to *parametric* architectural specifications i.e. those having lambda expressions as result, while in Section 5 we present a larger example motivating the introduction of the new rules, involving refinement of units with imports. Section 6 concludes the paper.

2 CASL Architectural Specifications

As mentioned above, CASL architectural specifications describe how the implementation is structured into component units. Each unit is given a name and assigned a specification; the intended meaning is to provide a model of the specification. Units can be *generic*, taking a list of specifications as arguments and having a result specification; such units denote partial functions that

take as arguments models of the parameter specifications and return a model of the result specification. The result is required to preserve the parameters (*persistency*), with the intuition that the program of the parameter must not be re-implemented, and the function is only defined on *compatible* models, meaning that the implementation of the parameters must be the same on common symbols. Units are combined in unit expressions with operations like renaming, hiding, amalgamation and applications of generic units. Again, terms are only defined for compatible models, in the sense that common symbols must be interpreted in the same way. Let us mention that architectural specifications are independent of the underlying formalism used for basic specifications, which is modelled as an institution [5].

An architectural specification consists of a list of unit definitions and declarations followed by a result unit expression. Fig. 1 presents a fragment of the grammar of the Casl architectural language that is relevant for the examples of this paper; the complete grammar can be found in [4]. Notice that we allow the specification of a unit to be itself architectural (named or anonymous) and that for units declarations there is an optional list of imported units (marked with $<_>$). The list must be empty when *USP* is architectural. Moreover, in Fig. 1 A is a unit name, S is a specification name, SP is a structured specification and σ is a signature morphism. We denote $\iota_{\Sigma \subseteq \Sigma'}$ the injection of Σ in Σ' when Σ' is a union of signatures with Σ among them.

$$
\begin{array}{l}
\mathit{ASP} ::= \textbf{units } \mathit{UDD}_1 \ldots \mathit{UDD}_n \\
\qquad\quad \textbf{result } \mathit{UE} \\
\mathit{UDD} ::= \mathit{UDEFN} \mid \mathit{UDECL} \\
\mathit{UDECL} ::= A : \mathit{USP} \ < \textbf{given } \mathit{UT}_1, \cdots \mathit{UT}_n > \\
\mathit{USP} ::= \mathit{SP} \mid \mathit{SP}_1 \times \cdots \times \mathit{SP}_n \to \mathit{SP} \mid \\
\qquad\quad \textbf{arch spec } S \mid \textbf{arch spec } \{\mathit{ASP}\} \\
\mathit{UDEFN} ::= A = \mathit{UE} \\
\mathit{UE} ::= \mathit{UT} \mid \lambda\ A_1 : \mathit{SP}_1, \ldots,\ A_n : \mathit{SP}_n \bullet \mathit{UT} \\
\mathit{UT} ::= A \mid A\ [\mathit{FIT}_1] \ldots [\mathit{FIT}_n] \mid \mathit{UT} \textbf{ and } \mathit{UT} \mid \mathit{UT} \textbf{ with } \sigma : \Sigma \to \Sigma' \mid \\
\qquad \mathit{UT} \textbf{ reduction } \sigma : \Sigma \to \Sigma' \mid \textbf{local } \mathit{UDEFN}_1 \ldots \mathit{UDEFN}_n \textbf{ within } \mathit{UT} \\
\mathit{FIT} ::= \mathit{UT} \mid \mathit{UT} \textbf{ fit } \sigma : \Sigma \to \Sigma'
\end{array}
$$

Fig. 1. Restricted language of architectural specifications.

The Casl semantics produces for any specification a signature and a class of models over that signature. This is not different for architectural specifications: the basic static semantics yields an architectural signature, while the model semantics produces an architectural model. We give definitions of this notions and a brief overview of the two semantics below.

An architectural signature consists of a unit signature for the result together with a static unit context, describing the signatures of each unit. A unit signature can be either a plain signature or a list of signatures for the arguments and a signature for the result. Starting with the initial empty static unit context, the static semantics for declarations and definitions adds to it the signature of each new unit and the static semantics for unit terms and expressions does the type-checking in the current static context. For any architectural specification ASP,

we denote $|ASP|$ the specification obtained by removing everything but the signature from the specifications used in declarations.

Model semantics is assumed to be run only after a successful run of the basic static semantics and it produces an architectural model over the resulting architectural signature. Model semantics of an individual unit is either simply a model of the specification, for non-generic units, or a partial function taking compatible models of the argument specifications to a model of the result specification. The result is required to protect the parameters when reduced back to a model of the corresponding signature. Generic units can be interpreted as total functions by introducing an additional value $\perp$ - this ensures consistency of generic unit specifications in $|ASP|$ whenever the unit specification is already consistent in an architectural specification ASP and is called *partial model semantics* in [4], Section IV:5. An architectural model over an architectural signature consists of a result unit over the result unit signature and a collection of units over the signatures given in the static context, named by their unit names. Model semantics produces a unit context, which is a class of unit environments - maps from unit names to units, and a unit evaluator, which is a map that gives a unit when given a unit environment in the unit context. The analysis starts with the unit context of all environments and each declaration and definition enlarges the unit context, adding a new constraint. Finally, the semantics of unit terms produces a unit evaluator for a given unit context.

$$\frac{\begin{array}{c} P_{st}(F) = \tau : \Sigma \to \Sigma' \\ C_{st} \vdash T \rhd \Sigma^A \\ \sigma : \Sigma \to \Sigma^A \\ (\sigma_R, \tau_R, \Sigma_R) \text{ is the pushout of } (\sigma, \tau) \end{array}}{P_{st}, C_{st} \vdash F[T \text{ fit } \sigma] \rhd \Sigma_R}$$

$$\frac{\begin{array}{c} C \vdash T \rhd UEv \\ \text{for each } E \in C, UEv(E)|_\sigma \in domE(F) \text{ } \mathbf{(i)} \\ \text{for each } E \in C, \text{ there is a unique } M \in Mod(\Sigma_R) \text{ such that} \\ M|_{\tau_R} = UEv(E) \text{ and } M|_{\sigma_R} = E(F)(UEv(E)|_\sigma) \text{ } \mathbf{(ii)} \\ UEv_R = \{E \mapsto M | E \in C, M|_{\tau_R} = UEv(E), M|_{\sigma_R} = E(F)(UEv(E)|_\sigma)\} \end{array}}{C \vdash F[T \text{ fit } \sigma] \rhd UEv_R}$$

Fig. 2. Basic static and model semantics rules for unit application

Fig. 2 presents the basic static semantics and model semantics rules for unit application (notice that we simplify to the case of units with just one argument). The static semantics rule produces the signature of the term T and returns as signature of $F[T]$ the pushout Σ_R of the span (σ, τ), where τ is the unit signature of F stored in the list of parameterized unit signatures P_{st}.

The model semantics rule first analyzes the argument T and gives a unit evaluator UEv. Then, provided that the conditions (i) the actual parameter actually fits the domain and (ii) the models $UEv(E)$ and $E(F)(UEv(E)|_\sigma)$ can be amalgamated to a Σ_R-model M hold, the result unit evaluator UEv_R gives the amalgamation M for each $E \in C$.

Typically one would expect that conditions (ii) would be discarded statically. For this purpose, an extended static semantics was introduced in [11], where the dependencies between units are tracked with the help of a diagram of signatures. The idea is that we can now verify that the interpretation of two symbols is the same by looking for a "common origin" in the diagram, i.e. a symbol which is mapped via some paths to both of them. We will present in this paper only the relevant rules of extended static semantics in Section 3. We are going to make use of the following notions. A *diagram* D is a functor from a small category to the category of signatures of the underlying institution. In the following, let D be a diagram. A family of models $\mathcal{M} = \{M_p\}_{p \in Nodes(D)}$ indexed by the nodes of D is *consistent with* D if for each node p of D, $M_p \in Mod(D(p))$ and for each edge $e : p \to q$, $M_p = M_q|_{D(e)}$. A *sink* α on a subset K of nodes consists of a signature Σ together with a family of morphisms $\{\alpha_p : D(p) \to \Sigma\}_{p \in K}$. We say that D *ensures amalgamability* along $\alpha = (\Sigma, \{\alpha_p : D(p) \to \Sigma\}_{p \in K})$ if for every model family $\mathcal{M}$ consistent with D there is a unique model $M \in Mod(\Sigma)$ such that for all $p \in K$, $M|_{\alpha_p} = M_p$.

The two semantics of architectural specifications are related by a soundness result [11]: if the extended semantics of an architectural specification is defined, then so is the basic semantics and the latter gives the same result. In [4], completeness is also proved for a simplified variant of the architectural language [1] and with a modified model semantics. We will discuss this in more detail in Section 4.1.

3 Semantics of Generic Unit Expressions

We present now the extended static semantic rule for generic unit expressions, with the help of a typical example of a dependency between the unit term of a lambda expression and the generic unit defined by it. Such dependencies are not tracked in the diagram built with the rules for extended static semantics defined in [4].

Example 1. Let us consider the Casl architectural specification from Fig. 3. The unit term $L1[A1]$ **and** $L2[A2]$ is ill-formed w.r.t. the rules of extended static semantics for architectural specifications because in the diagram in the Fig. 4 (built using the extended static semantics rules for generic unit expressions and unit applications, which are presented in Fig. 5 and Fig. 6 respectively) the sort s can not be traced to a common origin (which should be the node M).

□

The rule for analysis of generic unit expressions (Fig. 5) introduces a node p for the unit term of the lambda expression that keeps track of the sharing information of the terms involved. However, this node p is not further used

[1] It is nevertheless argued that the generalization to the full features of Casl architectural language is of no genuine complexity, excepting the case of imports. Our approach covers the imports as well.

spec S = **sort** s
spec S1 = **sort** $s1$
spec S2 = **sort** $s2$
arch spec ASP =
units M : S; A1 : S1; A2 : S2;
L1 = λ X1 : S1 $\bullet$ M **and** X1;
L2 = λ X2 : S2 $\bullet$ M **and** X2;
result L1 [A1] **and** L2 [A2]

Fig. 3. Lost sharing.

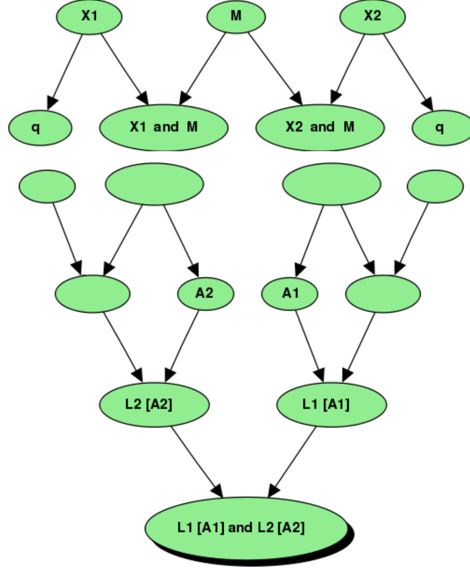

Fig. 4. Diagram of ASP.

$$\frac{\begin{array}{c}\Gamma_s \vdash \texttt{UNIT-BIND-1} \rhd (UN_1, \Sigma_1) \ldots \Gamma_s \vdash \texttt{UNIT-BIND-n} \rhd (UN_n, \Sigma_n) \\ \Sigma_a = \langle \Sigma_1, .., \Sigma_n \rangle \text{ and } \Sigma = \Sigma_1 \cup ... \cup \Sigma_n \\ UN_1, \ldots, UN_n \text{ are new names} \\ D' \text{ extends } dgm(C_S) \text{ by new node } q \text{ with } D'(q) = \Sigma, \\ \text{nodes } p_i \text{ and edges } e_i : p_i \to q \text{ with } D'(e_i) = \iota_{\Sigma_i \subseteq \Sigma} \text{ for } i \in 1, \ldots, n \\ C'_s = (\{\}, \{UN_1 \to p_1, \ldots, UN_n \to p_n\}, D') \\ \Gamma_s, C_s + C'_s \vdash \texttt{UNIT-TERM} \rhd (p, D'') \\ D'' \text{ ensures amalgamability along } (D''(p), \langle id_{D''(p)}, \iota_{\Sigma_i \subseteq D''(p)} \rangle_{i \in 1,\ldots,n}) \\ D''' \text{ extends } D'' \text{ by new node } z \text{ with } D'''(z) = \emptyset \end{array}}{\begin{array}{l}\Gamma_s, C_s \vdash \texttt{unit-expr UNIT-BIND-1,..,UNIT-BIND-n UNIT-TERM} \rhd \\ \qquad\qquad (z, \Sigma_a \to D''(p), D''')\end{array}}$$

Fig. 5. Extended static semantics rule for unit expressions (CASL Ref. Manual)

in application of lambda expressions. In the extended static context, the entry corresponding to the lambda expression only contains a new node labeled with the empty signature, denoted z in Fig. 5, as node of imports, and this new node is isolated. Notice also that the purpose of inserting the node q and the edges from nodes p_i to q is to ensure compatibility of the formal parameters when making the analysis of the unit term.

$$
\begin{array}{c}
C_s = (P_s, B_s, D) \\
P_s(UN) = (p^I, (\Sigma_1, ..., \Sigma_n \to \Sigma)) \\
\Sigma^F = D(p^I) \cup \Sigma_1 \cup ... \cup \Sigma_n \\
\Sigma_i, \Gamma_s, C_s \vdash \texttt{FIT-ARG-i} \rhd (\sigma_i : \Sigma_i \to \Sigma_i^A, p_i^A, D_i) \text{ for } i \in 1, \dots, n \\
D_1, .., D_n \text{ disjointly extend } D \\
\text{let } D^A = D_1 \cup .. \cup D_n \\
\Sigma^A = D(p^I) \cup \Sigma_1^A ... \cup \Sigma_n^A \\
\sigma^A = (id_{D(p^I)} \cup \sigma_1 \cup .. \cup \sigma_n) : \Sigma^F \to \Sigma^A \\
\sigma^A(\Delta) : \Sigma \to (\Sigma^A \cup \Sigma^A(\Delta)), \text{ where } \Delta : \Sigma^F \to \Sigma \text{ is the signature extension} \\
\Sigma^R = \Sigma^A \cup \Sigma^A(\Delta) \\
D^A \text{ ensures amalgamability along } (\Sigma^A, \langle \iota_{D(p^I) \subseteq \Sigma^A}, \iota_{\Sigma_i^A \subseteq \Sigma^A} \rangle_{i \in 1,\dots,n}) \\
D' \text{ extends } D^A \text{by new node } q^B, \text{ edge } e^I : p^I \to q^B \text{ with } D'(e^I) = \iota_{D(p^I \subseteq \Sigma)}, \\
\text{nodes } p_i^F \text{ and edges } e_i^F : p_i^F \to q^B \text{ with } D'(e_i^F) = \iota_{\Sigma_i \subseteq \Sigma} \\
\text{and } e_i : p_i^F \to p_i^A \text{ with } D'(e_i) = \sigma_i \text{ for } i \in 1, \dots, n \\
D' \text{ ensures amalgamability along } (\Sigma^R, \langle \sigma^A(\Delta), \iota_{\Sigma_i^A \subseteq \Sigma^R} \rangle_{i \in 1,\dots,n}) \\
D'' \text{ extends } D' \text{ by new node } q, \text{ edge } e' : q^B \to q \text{ with } D''(e') = \sigma^A(\Delta) \\
\text{and edges } e_i' : p_i^A \to q \text{ with } D''(e_i') = \iota_{\Sigma_i^A \subseteq \Sigma^R} \text{ for } i \in 1, \dots, n \\
\hline
\Gamma_s, C_s \vdash \texttt{unit-appl UN FIT-ARG-1,..,FIT-ARG-n} \ \rhd (q, D'')
\end{array}
$$

Fig. 6. Extended static semantics rules for unit application (CASL Ref. Manual)

Using this version of the rules raises a series of problems. First, there is no methodological justification for making terms like the one in our example illegal by not keeping track of the unit M in the lambda expressions. Moreover, ASP has a denotation w.r.t. the basic semantics (it is easy to see that the specification type-checks) and $|ASP|$ has a denotation w.r.t. the model semantics (there is no problem in amalgamating M with a model of specifications $S1$ or $S2$, since there are no shared symbols, and when making the amalgamation of $L1[A1]$ with L2[A2] the symbol s is interpreted in the same way by construction). Thus, since one would expect that the completeness result of [4] should still hold for the entire architectural language, ASP should have a denotation w.r.t. the extended static semantics.

Another reason to consider the current rules unsatisfactory is the relation between units with imports and generic units. A unit declaration with imports has been informally explained in the literature as a generic unit instantiated once, like in the following example.

Example 2. The following unit declarations, taken from the architectural specification of a steam boiler control system (Chapter 13 of [2]):

B : BASICS;
MR : VALUE → MESSAGES_RECEIVED **given** B;

can be expressed as a generic unit instantiated once (notice that the linear visibility of units, required in [4], is assumed to be extended):

B : BASICS;
MR : **arch spec** {
  units F : BASICS × VALUE → MESSAGES_RECEIVED
  result λ X : VALUE • F [B] [X]};

□

The two declarations in Example 2 are not equivalent because the former traces the dependency between MR and B while the latter does not. However it has been noticed that to be able to write down refinements of units with imports using the CASL refinement language designed in [8], this equivalence must become formal. This can only be the case if the second construction also tracks the dependency of B with MR.

Notice that in general the unit imported may be written as a more complex unit term and then its specification is no longer available directly. Moreover, as remarked in [7], it is not always possible to find a specification that captures exactly the class of all models that may arise as the result of the imported unit term. It is however possible to use the proof calculus for architectural specifications defined in [6] and Section IV.5.3 of [4] to generate a structured specification that includes this model class among its models. Another advantage of making the equivalence formal is that the completeness result for extended static semantics and the proof calculus for architectural specifications cover imports as well, since they can now be regarded only as "syntactic sugar" for the equivalent construction.

4 Adding Dependency Tracking

The proposed changes are based on the following observation: in the rule for unit application(Fig. 6), new nodes are needed for the formal parameters and for the result (labeled p_i^F and q^B respectively). However, for lambda expressions the nodes p_i and p in Fig. 5 have already been introduced with the same purpose. This symmetry can be exploited when making the applications of a lambda expression and we will therefore need to keep track of the mentioned nodes.

Recall that an extended static unit context consists of a triple (P_s, B_s, D), where $B_s \in UnitName \to Item$ and stores the corresponding nodes in the diagram for non-generic units, $P_s \in UnitName \to Item \times ParUnitSig$ and stores the parameterized unit signature of a generic unit together with the node of the imports, such that both B_s and P_s are finite maps and have disjoint domains and D is the signature diagram that stores the dependencies between units.

Firstly, we need to modify the definition of extended static unit contexts such that P_s maps now unit names to pairs in $[Item] \times ParUnitSig$, to be able to

store the nodes of the parameters and of the result for lambda expressions. Notice that a lambda expression must have at least one formal parameter, so the list of items contains either the node of the union of the imports in the case of generic units or at least two elements in the case of definitions of lambda expressions. Moreover, unit declarations of form UN : `arch spec` ASP where ASP is an architectural specification whose result unit is a lambda expression also should store the nodes for parameters and the result. The rule changes needed for this latter case are not straightforward and will be addressed separately in section 4.2. In Section 4.2 we will also make use of this list of nodes for a different purpose, namely tracking dependencies between different levels of visibility for units.

$$\frac{\begin{array}{c} \Gamma_s \vdash \texttt{UNIT-BIND-i} \rhd (UN_i, \Sigma_i) \text{ for } i \in 1, \ldots, n \\ \Sigma_a = \langle \Sigma_1, .., \Sigma_n \rangle \text{ and } \Sigma = \Sigma_1 \cup ... \cup \Sigma_n \\ UN_i \text{ are new names} \\ D' \text{ extends } dgm(C_S) \text{ by new node } q \text{ with } D'(q) = \Sigma, \\ \text{nodes } p_i \text{ with } D'(p_i) = \Sigma_i \\ \text{and edges } e_i : p_i \to q \text{ with } D'(e_i) = i_{\Sigma_i \subseteq \Sigma} \text{ for } i \in 1, \ldots, n \\ C'_s = (\{\}, \{UN_i \to p_i | i \in 1, \ldots, n\}, D') \\ \Gamma_s, C_s + C'_s \vdash \texttt{UNIT-TERM} \rhd (r, D'') \\ D'' \text{ ensures amalgamability along} (D''(r), \langle id_{D''(r)}, \iota_{\Sigma_i \subseteq D''(r)} \rangle) \\ \text{\sout{D''' extends D'' by new node z with $D'''(z) = \emptyset$}} \\ D''' \text{ removes from } D'' \text{ the node } q \text{ and its incoming edges} \end{array}}{\begin{array}{l} \Gamma_s, C_s \vdash \texttt{unit-expr UNIT-BIND-1,..,UNIT-BIND-n UNIT-TERM} \rhd \\ \qquad ([r, p_1, ..p_n], \Sigma_a \to D'''(r), D''') \end{array}}$$

Fig. 7. Modified extended static semantics rule for unit expressions.

Fig. 7 presents the modified static semantics rule for generic unit expressions, which introduces new nodes p_i for the parameters and a node q to ensure their compatibility during the analysis of the unit term. Then, the result node of the unit term p together with the nodes for parameters are returned as result of the analysis of the lambda expression, together with the diagram resulting by removing the node q and the edges from the nodes p_i to q from the diagram obtained after the analysis of the unit term. The reason why the node q must be removed is that the nodes of the formal parameters will be connected to the actual parameters and their compatibility must be rather checked than ensured.

We also have to make a case distinction in the rule of unit application. In the case of generic units, we can use the existing rule for unit applications. The rule for application of lambda expressions is similar with the one used in the first case, but it puts forward the idea that the nodes for formal parameters and result that were stored in the analysis of the lambda expression should be used when making the application. However, this requires special care, as we will illustrate with the help of some examples.

Example 3. Repeated applications of the same lambda expression. Let us consider the definition $F = \lambda X : SP \ . \ X$ **and** M where we assume that SP and the

specification of M do not share symbols and $M1, M2 : SP$. If we use the stored nodes for parameters and result at *every* application of F, we obtain the diagram in Fig. 8, resulting after applying F to $M1$ and $M2$. Notice that the edges from X to $M1$ and $M2$ respectively introduce a sharing requirement between the actual parameters, which is not intended. □

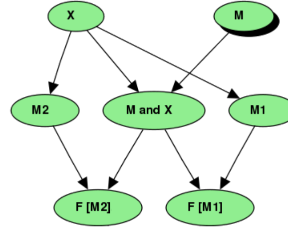

Fig. 8. Unwanted sharing.

The solution to this problem is to copy at every application the nodes introduced in the diagram during the analysis of the term of the lambda expression. The copy can be obtained starting with the stored nodes p_i by marking their copies as new formal parameter nodes and going along their outgoing edges: for each new node accessible from p_i, we introduce a copy of it in the diagram together with copies of its incoming edges - this last step copies also the dependencies of the unit term of the lambda expression with the outer units (in the example, the edge from the node of M to the node of M **and** X is copied). The copying stops when all nodes have been considered, and the copy of the result node is then marked as new result node. Let us denote the procedure described above *copyDiagram*, which takes as inputs the nodes for result and formal parameters of the lambda expression and the current diagram and returns the copied nodes for formal parameters and result and the new diagram. The procedure described works as expected because the diagram created during the analysis of the unit term of the lambda expression consists of exactly the nodes accesible from the formal parameter nodes and it has no cycles; moreover, no new dependencies involving these nodes are ever added in the diagram.

Example 4. Tracking dependencies of the actual parameters with the environment.

Let us consider the architectural specification in Fig. 9, where the actual parameter and the unit A used in the term of the lambda expression share the sort symbol s, which can be traced in the dependency diagram to a common origin, which is the node of P - see Fig. 10. This application should be therefore considered correct. □

Refering to the rule in Fig. 6, the generic unit is given by the inclusion $\Delta : \Sigma^F \to \Sigma$ of its formal parameters into the body and at application, the fitting arguments give a signature morphism $\sigma^A : \Sigma^F \to \Sigma^A$ from the formal parameters to the actual parameters. Then, $\Sigma^A \cup \Sigma^A(\Delta)$ results by making the union of the fitting arguments with the body translated along the signature extension $\sigma^A(\Delta) : \Sigma \to \Sigma^A \cup \Sigma^A(\Delta)$. Originally, an application has been considered not well-formed if the result signature is not a pushout of the body and argument signatures (this is hidden in the use of the notation $\sigma^A(\Delta)$, see [4]) and notice that this is indeed not the case in Example 4. We can drop this requirement in the case of lambda expressions and rely on the condition that the diagram should ensure amalgamability; indeed, in this case the application is correct if

spec S = **sort** x
spec T = **sorts** s, t
spec U = **sorts** s, u
arch spec ASP =
  units
  P : {**sort** s};
  A : T **given** P;
  L = λ X : S • A **and** X;
  B : U **given** P
  result L [B **fit** $x \mapsto u$]

Fig. 9. Sharing between arguments and actual parameter.

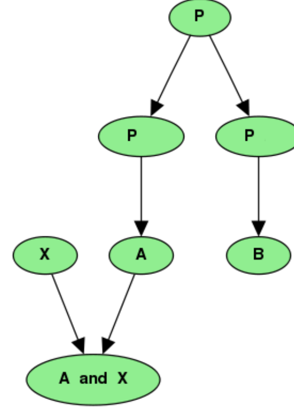

Fig. 10. Diagram of Example 4 before application.

whenever a symbol is present both in the body and in the argument signatures, the symbol can be traced in the diagram to a common origin which need not be the node of the formal parameter, like in the case of sort s above.

Taking into account the observations in Examples 3 and 4, the rule of for application of lambda expressions is presented in Fig. 11.

$$\frac{\begin{array}{c} C_s = (P_s, B_s, L_s, D_0) \\ L_s(UN) = ([p, p_1, \ldots, p_n], (\Sigma_1, \ldots, \Sigma_n \to \Sigma)) \\ ([r, f_1, \ldots, f_n], D) = \mathit{copyDiagram}([p, p_1, \ldots, p_n], D_0) \\ \Sigma^F = \Sigma_1 \cup ... \cup \Sigma_n \\ \Sigma_i, \Gamma_s, C_s \vdash \texttt{FIT-ARG-i} \rhd (\sigma_i : \Sigma_i \to \Sigma_i^A, p_i^A, D_i) \text{ for } i \in 1, \ldots, n \\ D_1, .., D_n \text{ disjointly extend } D \\ \text{let } D^A = D_1 \cup .. \cup D_n \\ \Sigma^A = \Sigma_1^A ... \cup \Sigma_n^A \\ \sigma^A = (\sigma_1 \cup .. \cup \sigma_n) : \Sigma^F \to \Sigma^A \\ \sigma^A(\Delta) : \Sigma \to (\Sigma^A \cup \Sigma^A(\Delta)), \text{ where } \Delta : \Sigma^F \to \Sigma \text{ is the signature extension} \\ \text{and the pushout condition for } \Sigma^A \cup \Sigma^A(\Delta) \text{ is dropped} \\ \Sigma^R = \Sigma^A \cup \Sigma^A(\Delta) \\ D^A \text{ ensures amalgamability along } (\Sigma^A, \langle \iota_{\Sigma_i^A \subseteq \Sigma^A} \rangle_{i \in 1,\ldots,n}) \\ D' \text{ extends } D^A \text{ with edges } e_i : f_i \to p_i^A \text{ with } D'(e_i) = \sigma_i, \text{ for } i \in 1, \ldots, n \\ D' \text{ ensures amalgamability along } (\Sigma^R, \langle \sigma^A(\Delta), \iota_{\Sigma_i^A \subseteq \Sigma^R} \rangle_{i \in 1,\ldots,n}) \\ D'' \text{ extends } D' \text{ by new node } q, \text{ edge } e' : r \to q \text{ with } D''(e') = \sigma^A(\Delta) \\ \text{and edges } e'_i : p_i^A \to q \text{ with } D''(e'_i) = \iota_{\Sigma_i^A \subseteq \Sigma^R}, \text{ for } i \in 1, \ldots, n \end{array}}{\Gamma_s, C_s \vdash \texttt{unit-appl UN FIT-ARG-1,..,FIT-ARG-n} \rhd (q, D'')}$$

Fig. 11. Extended static semantics rule for unit application of lambda expressions.

Fig. 12 presents the diagram of the architectural specification ASP in Example 1 using the modified rules of Fig. 7 and 11[2]; notice that in this diagram the sort s can be traced to a common origin and thus the amalgamation is correct. Moreover, when making the application of the lambda expression, the diagram of the term M **and** X is copied such that no dependency between the actual parameters is incorrectly introduced by edges from the formal parameter node and copying the diagram does not duplicate the node M.

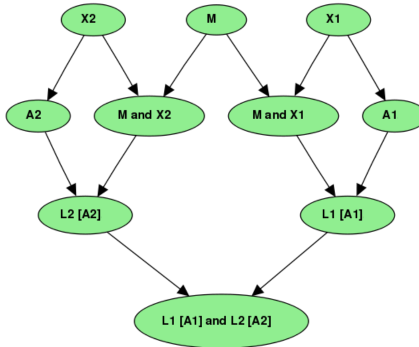

Fig. 12. Diagram of Example 1 with the new rules.

4.1 Completeness of Extended Static Semantics

In this section we will extend the soundness and completeness result from [4] to the architectural specification language obtained by adding definitions of generic units to the original fragment language in Section IV.5 of [4], i.e. unit definitions assign to unit names unit expressions instead of unit terms. Comparing with the language in Fig. 1, the differences are that this language does not mix declarations and definitions of units, i.e. all declarations are done locally in the **local** ... **within** construction, unit declarations do not have imports and unit specifications are never architectural. Also we only restrict to lambda expressions with a single parameter. Notice that these differences do not modify the language in an essential way. The soundness and completeness result is formulated as follows.

Theorem 1. *For any architectural specification ASP in which no generic unit is applied more than once we have that ASP has a denotation w.r.t. the extended static semantics iff ASP has a denotation w.r.t. the static semantics and* $|ASP|$ *has a denotation w.r.t. the partial model semantics.*

---

[2] Note that we omitted the nodes of the term of the lambda expression that are copied at each application and only kept the significant ones.

The requirement that no generic unit is applied more than once is a simplifying assumption for achieving a *generative* semantics, i.e. repeated applications of a generic unit to same arguments no longer yields the same result.

The theorem is proved using a quite technical lemma (Lemma 5.6 in [4]) which we don't present in full detail here. Intuitively, it says that the extended static semantics for a unit term is successful if and only if the static and model semantics are successful as well and if it is the case, the signatures match and the environment obtained in the model semantics can be represented as a family of models compatible with the diagram obtained in the extended static semantics. The proof of this lemma is done by induction on the structure of the unit term. In order to extend the proof to cover lambda expressions as well, we have two new cases to consider: applications of lambda expressions and local declarations of generic units. The new proof is quite long and tedious, but follows very closely the existing proof. Therefore, we only sketch here the proof idea. For applications of lambda expressions, we simply repeat the proof for unit applications but use this time the copies of the nodes for arguments and result that are stored in the context instead of introducing arbitrary distinguished ones. For local declarations of generic units, the proof is similar to the one of local declarations of non-generic units, only that now we have to spell out the rules for lambda expressions before applying the inductive step for the unit term in the lambda expression. The introduced dependency between the lambda expression and its unit term is essential when proving compatibility of the environment with the diagram.

4.2 Parametric Architectural Specifications

Further changes are needed when considering the complete language in Fig. 1. The result unit of an architectural specification ASP can be itself a lambda expression. In this case the architectural specification is called parametric. Notice that the grammar of the architectural language also covers the case when the specification of a unit is itself architectural (either named or anonymous). For such units, we must ensure that designated nodes for formal parameters and result exist in the diagram, since they are required in the rule of unit application of generic units.

Let us first consider the case of anonymous parametric architectural specifications. For the specification below, the static analysis of the architectural specification is currently done in the empty extended static context and thus the nodes for formal parameters and result, which are introduced when making the analysis of the result lambda expression, are no longer present in the diagram at the global level. Notice that the dependency between M and F must be tracked in the diagram in order to ensure correctness of the term $F[M1$ **fit** $t \mapsto u]$ **and** $F[M2$ **fit** $t \mapsto v]$.

spec S = **sort** s
spec T = **sort** t
spec U = **sort** u
spec V = **sort** v

arch spec ASP2 =
  units
  F : **arch spec** {
    units M : S
    result λ X : T • M **and** X
    };
  M1 : U; M2 : V;
  result F [M1 **fit** $t \mapsto u$] **and** F [M2 **fit** $t \mapsto v$]

The way we overcome this problem is by making the analysis of the inner architectural specification in the existing global context instead of using an empty global context. After the analysis, we will keep in the global context the diagram resulting from the analysis of the locally-declared units. Thus, the nodes introduced locally become available for further references. Moreover, the units declared locally will not be kept in the global extended context, since we do not want to extend their scope. By making the analysis of the local specification in the global context, the visibility of units declared at the global level is extended to the local context as well (remember that we assumed this extension of visibility in Example 2) and the dependencies of the global units with the local environment are tracked by keeping the entire resulting diagram at the global level.

The second case to consider is the one of unit declarations of form U : **arch spec** ASP, when ASP is a named parametric architectural specification. In this case, ASP cannot refer to units other than those declared within itself and therefore its diagram does not carry any dependency information relevant for the global level. Therefore, instead of adding the diagram of ASP to the global diagram, we only need to introduce new nodes for formal parameters and edges to a new result node. This abstracts away the dependencies of the result node of ASP with the units declared locally (which we don't need) and only keeps the dependencies of the result node with the parameter nodes along the new edges, which will be then copied as diagram of the unit term of the lambda expression at each application of U.

The modifications of the extended static semantics rules are presented in figures 13 to 21 and can be summarized as follows. At the library level, the analysis of an architectural specification (Fig. 13) starts in the empty extended static unit context. The analysis of an architectural specification (Fig. 14), we need to extend the diagram for anonymous parametric architectural specifications (first rule) and named parametric architectural specifications (third rule). In the latter case, we also need to return the (new) nodes for formal parameters and result $(r, p_1, \cdots, p_n)$. The rule for basic architectural specifications (Fig. 15) analyzes the list of declarations and definitions in the context received as parameter rather than in the empty context like before. Thus the diagrams built locally will be added to the global diagram and the visibility of global units is extended. The rule for result unit (Fig. 17) makes a case distinction for each of the four alternatives in Fig. 1. When the specification of the unit is not architectural (first two rules), the imported units are analyzed, a new node p labelled

with the signature union of all imports is introduced in the diagram and the dependency between the declared unit and the imports is tracked either via the edge from p to q in the first case, or by storing the node p as node of imports in the second case. When the specification of the unit is a parametric architectural specification (third rule), the nodes of formal parameters and results are saved and the unit will be applied using the rule for lambda expressions. Finally, when the specification of the unit is a non-parametric architectural specification (last rule), we set the pointer for the unit to the node of the result unit of the architectural specification to be able to trace its dependencies. Notice that in the last two cases there are no imports so the node p will always be labeled with the empty signature. The changes made for unit specifications (Figures 18 to 20) are just meant to propagate the results.

$$\frac{\begin{array}{c}\Gamma_S = (G_s, V_s, A_s, T_s) \\ ASN \text{ is a new name} \\ \Gamma_s, C_0 \vdash \texttt{ARCH-SPEC} \rhd (nodes, A\Sigma, D')\end{array}}{\Gamma_S \vdash \texttt{arch-spec-defn}\ ASN\ \texttt{ARCH-SPEC} \rhd (G_s, V_s, A_s \cup \{ASN \mapsto A\Sigma\}, T_s)}$$

Fig. 13. Rule for architectural library items.

$$\boxed{\Gamma_s, C_s \vdash \texttt{ARCH-SPEC} \rhd (nodes, A\Sigma, D)}$$

$$\frac{\Gamma_s, C_s \vdash \texttt{BASIC-ARCH-SPEC} \rhd (nodes, A\Sigma, D')}{\Gamma_s, C_s \vdash \texttt{BASIC-ARCH-SPEC}\ \text{qua}\ \texttt{ARCH-SPEC} \rhd (nodes, A\Sigma, D')}$$

$$\frac{\begin{array}{c}ASN \in Dom(A_s) \\ A_s(ASN) = (S, \Sigma) \\ D' \text{ extends } dgm(C_s) \text{ with a new node } n \text{ such that } D'(n) = \Sigma\end{array}}{(G_s, V_s, A_s, T_s), C_s \vdash ASN\ \text{qua}\ \texttt{ARCH-SPEC} \rhd ([n], A_s(ASN), dgm(C_s))}$$

$$\frac{\begin{array}{c}ASN \in Dom(A_s) \\ A_s(ASN) = (S, \langle \Sigma_1, ..., \Sigma_n \rangle \to \Sigma) \\ D' \text{ extends } dgm(C_s) \text{ with new nodes } p_1, .., p_n, r \text{ and edges } p_i \to r \\ \text{such that } D'(p_i \to r) = \iota_{\Sigma_i \subseteq \Sigma}\end{array}}{(G_s, V_s, A_s, T_s), C_s \vdash ASN\ \text{qua}\ \texttt{ARCH-SPEC} \rhd ([r, p_1, ..., p_n], A_s(ASN), D')}$$

Fig. 14. Rules for architectural specifications.

$$\boxed{\Gamma_s, C_s \vdash \texttt{BASIC-ARCH-SPEC} \rhd (nodes, A\Sigma, D)}$$

$$\frac{\begin{array}{c}\Gamma_s, C_s^0 \vdash UDD^+ \rhd C_s \\ \Gamma_s, C_s \vdash \texttt{RESULT-UNIT} \rhd (nodes, U\Sigma, D)\end{array}}{\Gamma_s, C_s^0 \vdash \texttt{basic-arch-spec}\ UDD^+\ \texttt{RESULT-UNIT} \rhd (nodes, (ctx(C_s), U\Sigma), D)}$$

Fig. 15. New extended static semantics rule for basic architectural specifications.

$$\boxed{\Gamma_s, C_s \vdash \texttt{UNIT-DECL-DEFN}^+ \rhd C'_s}$$

$$\frac{\begin{array}{c}\Gamma_s, C_s^0 \vdash \texttt{UDD1} \rhd (C_s)_1 \\ \ldots \\ \Gamma_s, (C_s)_{n-1} \vdash \texttt{UDDn} \rhd (C_s)_n\end{array}}{\Gamma_s, C_s^0 \vdash \texttt{UDD1, ..., UDDn} \rhd (C_s)_n}$$

Fig. 16. New extended static semantics rule for lists of declarations and definitions.

$$\boxed{\Gamma_s, C_s \vdash \texttt{RESULT-UNIT} \rhd (nodes, U\Sigma, D)}$$

$$\frac{\Gamma_S, C_s \vdash \texttt{UNIT-EXPR} \rhd (p, U\Sigma, D)}{\Gamma_s, C_s \vdash \texttt{result-unit UNIT-EXPR} \rhd ([p], U\Sigma, D)}$$

$$\frac{\Gamma_S, C_s \vdash \texttt{UNIT-EXPR} \rhd (r : fs, U\Sigma, D)}{\Gamma_s, C_s \vdash \texttt{result-unit UNIT-EXPR} \rhd (r : fs, U\Sigma, D)}$$

Fig. 17. New extended static semantics rule for result unit expressions.

$$\boxed{\Gamma_S, C_s \vdash \texttt{ARCH-UNIT-SPEC} \rhd (nodes, U\Sigma, D)}$$

$$\frac{\Gamma_s, C_s \vdash \texttt{ARCH-SPEC} \rhd (nodes, (S, U\Sigma), D')}{\Gamma_s, C_s \vdash \texttt{ARCH-SPEC}\ \text{qua}\ \texttt{ARCH-UNIT-SPEC} \rhd (nodes, U\Sigma, D')}$$

Fig. 18. New extended static semantics rule for architectural unit specifications

$$\boxed{\Gamma_s, C_s \vdash \texttt{unit-defn}\ UN\ \texttt{UNIT-EXPR} \rhd C'_s}$$

$$\frac{\begin{array}{c}\Gamma_s, C_s \vdash \texttt{UNIT-EXPR} \rhd ([p], \Sigma, D) \\ UN \text{ is a new name}\end{array}}{\Gamma_s, C_s \vdash \texttt{unit-defn}\ UN\ \texttt{UNIT-EXPR} \rhd (\{\}, \{UN \mapsto (p, \Sigma)\}, D)}$$

$$\frac{\begin{array}{c}\Gamma_s, C_s \vdash \texttt{UNIT-EXPR} \rhd (r : fs, U\Sigma, D) \\ UN \text{ is a new name}\end{array}}{\Gamma_s, C_s \vdash \texttt{unit-defn}\ UN\ \texttt{UNIT-EXPR} \rhd (\{UN \mapsto (r : fs, U\Sigma)\}, \{\}, D)}$$

Fig. 19. New rule for unit definitions.

$$\boxed{\Sigma, \Gamma_s, C_s \vdash \texttt{UNIT-SPEC} \rhd (nodes, U\Sigma, D)}$$

$$\frac{\Gamma_s, C_s \vdash \texttt{ARCH-UNIT-SPEC} \rhd (nodes, U\Sigma, D')}{\Sigma, \Gamma_s, C_s \vdash \texttt{ARCH-UNIT-SPEC}\ \text{qua}\ \texttt{UNIT-SPEC} \rhd (nodes, U\Sigma, D')}$$

Fig. 20. New extended static semantics rule for arch unit specs as unit specs.

5 An Application: Refinement of Units with Imports

This section illustrates the use of the new semantics rules for architectural specifications with the help of a case study example - the specification of a warehouse system by Baumeister and Bert [1]. The system keeps track of stocks of products and of orders and allows adding, canceling and invoicing orders, as well as adding products to the stock.

$$\boxed{\Gamma_s, C_s \vdash \texttt{UNIT-DECL} \rhd (C'_s, D)}$$

$$\frac{\begin{array}{c} C_s \vdash \texttt{UNIT-IMPORTED} \rhd (p, D) \\ C = C_s + (\{\}, \{\}, D) \\ D(p), \Gamma_s, C \vdash \texttt{UNIT-SPEC} \rhd ([], \Sigma, D') \\ UN \text{ is a new name} \\ D'' \text{ extends } D' \text{ by a new node } q \text{ with } D''(q) = D'(p) \cup \Sigma \\ \text{and edge } e : p \to q \text{ with } D''(e) = \iota_{D'(p) \subseteq D''(q)} \end{array}}{\Gamma_s, C_s \vdash \texttt{unit-decl } UN \texttt{ UNIT-SPEC UNIT-IMPORTED} \rhd (\{\}, \{UN \mapsto q\}, D'')}$$

$$\frac{\begin{array}{c} C_s \vdash \texttt{UNIT-IMPORTED} \rhd (p, D) \\ C = C_s + (\{\}, \{\}, D) \\ D(p), \Gamma_s, C \vdash \texttt{UNIT-SPEC} \rhd ([], \langle \Sigma_1, .., \Sigma_n \rangle \to \Sigma_0, D') \\ UN \text{ is a new name} \end{array}}{\begin{array}{l} \Gamma_s, C_s \vdash \texttt{unit-decl } UN \texttt{ UNIT-SPEC UNIT-IMPORTED} \rhd \\ \qquad (\{UN \mapsto (p, \langle \Sigma_1, .., \Sigma_n \rangle \to \Sigma_0 \cup \Sigma^I)\}, \{\}, D') \end{array}}$$

$$\frac{\begin{array}{c} C_s \vdash \texttt{UNIT-IMPORTED} \rhd (p, D) \\ C = C_s + (\{\}, \{\}, D) \\ D(p), \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd (r : fp, \langle \Sigma_1, .., \Sigma_n \rangle \to \Sigma, D') \\ UN \text{ is a new name} \end{array}}{\begin{array}{l} \Gamma_s, C_s \vdash \texttt{unit-decl } UN \texttt{ UNIT-SPEC UNIT-IMPORTED} \rhd \\ \qquad (\{UN \mapsto (r : fp, \langle \Sigma_1, .., \Sigma_n \rangle \to \Sigma)\}, \{\}, D') \end{array}}$$

$$\frac{\begin{array}{c} C_s \vdash \texttt{UNIT-IMPORTED} \rhd (p, D) \\ C = C_s + (\{\}, \{\}, D) \\ D(p), \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd ([n], \Sigma, D') \\ UN \text{ is a new name} \end{array}}{\begin{array}{l} \Gamma_s, C_s \vdash \texttt{unit-decl } UN \texttt{ UNIT-SPEC UNIT-IMPORTED} \rhd \\ \qquad (\{\}, \{UN \mapsto ([n], \Sigma)\}, D') \end{array}}$$

Fig. 21. New rules for unit declarations.

Fig. 22 presents the specifications involved and the relations between them. The specifications ORDER, PRODUCT and STOCK specify the objects of the system. The main purpose for the INVOICE specification is to specify an operation for invoicing an order for a product in the stock. The QUEUES and ORDER_QUEUES specifications specify different types of queues (pending, invoiced) for orders. The WHS specification is the top-level specification, with the main operations of the system. The next step is to come up with a more concrete realization of ORDER, that allows to distinguish between different orders on the same quantity of a product by introducing labels. This results in specifications ORDER', INVOICE' and WHS'. The specification WHS' of the warehouse system is then further refined to an architectural specification describing the structure of the implementation of the system. Moreover, NAT and LIST are the usual specifications of natural numbers and lists.

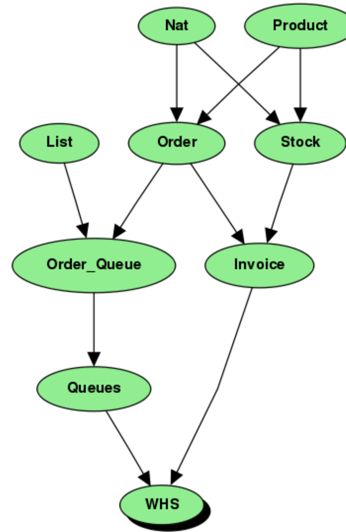

Fig. 22. Structure of the specification of the warehouse system.

The modular decomposition of the warehouse system is recorded in the architectural specification below:

arch spec WAREHOUSE =
  units NATALG : NAT; PRODUCTALG : PRODUCT;
  ORDERFUN : PRODUCT → ORDER' **given** NATALG;
  ORDERALG = ORDERFUN [PRODUCTALG];
  STOCKFUN : PRODUCT → STOCK **given** NATALG;
  STOCKALG = STOCKFUN [PRODUCTALG];
  INVOICEFUN : {ORDER' **and** STOCK} → INVOICE';
  QUEUESFUN : ORDER → QUEUES;
  WHSFUN : {QUEUES **and** INVOICE'} → WHS'
  result WHSFUN[QUEUESFUN [ORDERALG]
      and INVOICEFUN [ORDERALG **and** STOCKALG]]

Using the refinement language introduced in [8], we can write this refinement chain in the following way:

refinement R =
    WHS **refined to**
    WHS' **refined to arch spec** WAREHOUSE

We can further proceed to refine each component separately. For example, let us assume we want to further refine ORDER' in such a way that the labels of orders are natural numbers and denote the corresponding specification ORDER".

The changes in the extended static semantics rules allow us to rephrase the declaration of OrderFun in an equivalent way using generic units[3]:

OrderFun :
  arch spec
  {**units** F : NAT × PRODUCT → ORDER'
  result lambda X : PRODUCT • F [NatAlg] [X]
  };

Then we need to write a unit specification for the specification of OrderFun to be able to further refine it:

unit spec NATORDER' = NAT × PRODUCT → ORDER'

and another unit specification to store the signature after refinement as well:

unit spec NATORDER" = NAT × PRODUCT → ORDER"

The refinement is done along a morphism that maps the sort *Label* to Nat:

refinement R' =
  NATORDER' **refined via** *Label* ↦ *Nat* **to** NATORDER"

The Casl refinement language can be easily modified to allow the refinement of OrderFun without making use of the arbitrary name (in our case F) chosen for the generic unit [4]:

refinement R" = R **then** {OrderFun **to** R'}

6 Conclusions

We have presented and discussed a series of changes to extended static semantics of Casl architectural specifications, motivated by the unsatisfactory treatment of lambda expressions in the original semantics of Casl [4]. We have identified a number of practically important situations requiring lambda expressions to have dependency tracking with their unit term and we formulated the modified rules accordingly. We have also discussed briefly how the known completeness result can now be successfully extended to the whole Casl architectural language; a full proof is very lengthy and follows the lines of the existing result; for this reason we have omitted it. Finally, we have presented an example of refinement of generic units with imports; without the changes introduced in this paper such a refinement could not have been expressed using the Casl refinement language. The implementation of the modified rules in the Heterogeneous Tool Set Hets [9] is currently in progress.

---

[3] Notice that this equivalence becomes visible at the level of *refinement signatures* as defined in [8].

[4] More exactly, the composition of refinement signatures must be slightly adapted to make this composition legal.

Acknowledgments. I would like to thank Till Mossakowski and Lutz Schröder for prompt and detailed comments. I am grateful to Andrzej Tarlecki for suggesting a series of significant technical improvements. This work has been supported by the German Research Council (DFG) under grant MO-2428/9-1.

References

1. Hubert Baumeister and Didier Bert. Algebraic specification in CASL. In M. Frappier and H. Habrias, editors, *Software Specification Methods: An Overview Using a Case Study*, FACIT (Formal Approaches to Computing and Information Technology), chapter 12, pages 209–224. Springer, 2000.
2. Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004.
3. Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002.
4. Peter D. Mosses (Ed.). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.
5. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992.
6. Piotr Hoffman. Verifying generative CASL architectural specifications. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, pages 233–252. Springer, 2003.
7. Piotr Hoffman. *Architectural Specifications and Their Verification*. PhD thesis, Warsaw University, 2005.
8. T. Mossakowski, D. Sannella, and A. Tarlecki. A simple refinement language for CASL. In Jose Luiz Fiadeiro, editor, *WADT 2004*, volume 3423 of *LNCS*, pages 162–185. Springer; Berlin, 2005.
9. Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Orna Grumberg and Michael Huth, editors, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer-Verlag Heidelberg, 2007.
10. D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica*, 25:233–281, 1988.
11. Lutz Schröder, Till Mossakowski, Piotr Hoffman, Bartek Klin, and Andrzej Tarlecki. Semantics of architectural specifications in CASL. In *Fundamental Approaches to Software Engineering*, volume 2029 of *LNCS*, pages 253–268. Springer;, 2001.