CIF MSOS type system

Citation for published version (APA):

Document status and date:
Published: 01/01/2012

Publisher Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 02. Oct. 2020
CIF MSOS type system

A.P. van der Meer

Computer Science Reports 12-06
Eindhoven, February 2012
CIF MSOS type system

A.P. van der Meer

February 6, 2012

1 Introduction

This document provides a formal description of the type system of a modeling language called Compositional Interchange Format (CIF) [6, 7]. CIF can be used to create and describe specifications of systems containing components that evolve continuously in time, and components that do discrete steps. Such systems often occur when a discrete (electronic) controller, like a thermostat, has to control a continuous (physical) system, like a heating system. The temperature in the object or room to be heated will change in a continuous manner, but the controller can only react to these changes in a discrete manner, for example switching an actuator on or off. By combining these two kinds of systems in one specification, more complete description of the behavior is obtained, and analysis of their combined behavior can be done [5].

In order to perform this analysis, we need to know what a given CIF specification means. This meaning is known as the semantics of the specification. The details of the behavior of systems define by so-called dynamic semantics describe the behavior of a CIF specification when given input is applied. In contrast, so-called static semantics define properties of models that can be determined without considering input. A type system is a component of the static semantics and defines classification of model elements that produce values, usually called expressions, by assigning types. The process of assigning types is colloquially referred to as “typing”. In theory, we can apply a type system to a model without changing any model element, but in the CIF case, we want to retain the types calculated. The definition of CIF actually requires that elements representing types are present wherever possible in the model. To represent the model before it has been typed, a separate model definition, called Untyped CIF or UCIF, has been created. We describe the type system of CIF by relating UCIF model elements to CIF model elements. In order to create valid a CIF model, elements representing types are added to represent the computed types.

Type systems are usually implemented via type checkers, to determine whether for a given model all parts that need to be typed according to the type system can be typed. If this is not the case, errors are present in the model. The errors we are interested in, occur because certain combinations of constructs and their properties have no defined semantics. For example, most languages define so-called operations that combine the values that result from subexpressions. The dynamic semantics of the language define what the results are for each operation for given values. For example, an operation representing addition might be defined such that when given two subexpressions that result in the value “1”, it produces the value “2” [35]. Most operations cannot combine arbitrary values, but only values of specific types. If the subexpressions of an operation produce values the operation is not defined for, for example when the aforementioned addition operation is given the values “1” and “true”, this disrupts the semantics of the whole CIF model that contains this expression. The dynamic semantics simply do not specify what happens when an operation cannot produce a value when applied. This means models that
lack dynamic semantics cannot be used for simulation or transformation because the results will either be nonexistent or arbitrary. Thus we want to identify these models and if possible, indicate where problems may be. The goal of this document is to provide a baseline formalization of the type system of CIF, that can be used as the basis for the implementation of a type checker.

In this document, we will first describe CIF in more detail in Section 2. Then we will describe the background of the formalism used to define the type system, MSOS, in Section 3. Section 4 contains a number of preliminary matters such as notation details and function definitions. The next chapter, Section 5, contains an example of how the type system can be applied to model elements. The actual type system is described in the next two Sections, 6 and 7, which are followed by related work in Section 8 and conclusions and discussion of future work in Section 9.

2 CIF

CIF is a modeling language intended to be a bridge between various tools for modeling discrete, continuous and hybrid systems, i.e. systems modeling both discrete and continuous behavior. Additionally, it can be used to create new models of hybrid systems directly. In principle, the structure of CIF models is defined in a metamodel that will be introduced in Section 2.1. Actual CIF models are stored in the so-called ECORE format [31]. This XML format is designed to be easy to use in automated transformation tools. However, the format is verbose [12] and inconvenient for humans to read and write [32]. For modeling, a textual representation has been created for CIF models. Figure 2, taken from [4], shows an example of a hybrid system modeled in CIF in its textual form. The example specification describes the behavior of a controller of the level of water in a tank, as shown in Figure 1. The model has three parts. The first part, consisting of lines 2 to 4, is where the variables of the model are declared and initialized. The second part, at lines 5 to 10, is where the behavior of the continuous part of the model is described. Finally, the third part, at lines 12 to 16, is where the behavior of the discrete part is described. The latter two parts are executed in parallel, as indicated by the CIF parallelism operator, | |, at line 11. The current volume of the contents of the tank is described by \( V \), a continuous variable. The tank has an inflow controlled by a valve that can be switched on or off, with a rate described by the variable \( Q_i \). The state of the valve is described by the discrete variable \( n \). The outflow of the tank, described by \( Q_o \), cannot be controlled, but is dependent only on physics.

The behavior of the discrete controller part of the system is described in the third part. The function of this controller is to keep the volume of the tank between certain limits. The controller is defined in the form of a state machine with two states, closed, representing that the controller has decided to close the valve and described in line 14, and opened, representing that the controller has decided to open the valve and described in line 15. The state changes whenever the volume in the tank reaches a threshold value, and the valve is opened or closed accordingly.

The behavior of the discrete controller part of the system is described in the third part. The function of this controller is to keep the volume of the tank between certain limits. The controller is defined in the form of a state machine with two states, closed, representing that the controller has decided to close the valve and described in line 14, and opened, representing that the controller has decided to open the valve and described in line 15. The state changes whenever the volume in the tank reaches a threshold value, and the valve is opened or closed accordingly.

In this model, we can see several instances of elements where the type system needs to make decisions. For example, operators and functions in modeling languages can have multiple implementations, depending on the types of the arguments, because some cases can be executed more efficiently. In the expression that defines \( Q_i \), shown in line 6, the type system needs to select the appropriate implementation of the \( * \) operator, based on the types of the subexpressions, and
then determine whether the result can be placed in $Q_i$.

2.1 Metamodels of CIF and UCIF

In this section we will introduce key parts of the metamodels of the Compositional Interchange Format (CIF) and its untyped form UCIF. The structure of both CIF and UCIF models are described by EMF metamodels. In EMF modeling [31], a metamodel is a special model that defines the structure of other models. To this end the metamodel in turn contains a number of elements that each define the structure of a class of model elements. Each model element is described by a specific class, called an EClass, in the metamodel. Each EClass has a name and a number of properties, that describe what data can be stored in an element. For example, if we look at the EClass \texttt{BoolLiteral}, shown at the bottom of Figure 3, we can see that the EClass has a name, \texttt{BoolLiteral}, and a property, \texttt{value}.

Both metamodels consist of one main package, called \texttt{cif} and \texttt{ucif} respectively, and three sub-packages, called \texttt{annotations}, \texttt{types} and \texttt{expressions}. Due to the complexity of the CIF language, the complete metamodel is very large, containing more than 100 EClasses. We restrict ourselves here to the two parts that are most relevant for the type system: the \texttt{expressions} package and the \texttt{types} package. The \texttt{expressions} package contains all elements for which the dynamic semantics produce values, thus they are exactly the elements that need to be typed. The \texttt{types} package contains all elements representing types, which will be used to store the computed values in the final model. Because the metamodels of UCIF and CIF are very similar, we will mainly discuss the CIF metamodel, and only describe the UCIF metamodel where it deviates from the CIF model.

2.1.1 UCIF & CIF Expressions Metamodel

The \texttt{expressions} package of the CIF metamodel contains, as one would expect, all classes related to CIF expressions. A graphical representation of the \texttt{expressions} package is shown in Figure 3. In this figure, each class is represented by a rounded box with the class name. Properties are represented in two ways. So-called \texttt{attributes}, properties that contain basic data, are listed in the
class box, under the class name. An example is the value attribute of the `BooleanLiteral` class, which indicates what value the literal represents. `References`, properties that link elements to other model elements are shown as arrows with small, black arrowheads pointing to the class that represents elements that can be linked to. The name of the reference is placed on the arrow. For example, the class `BinaryExpression` contains two references, called `leftChild` and `rightChild`. If the arrow has a black diamond at its base, the reference is called a `containment reference`. Containment references are used for situations where one element is considered part of another element, like a car engine is part of, and contained in, a car. As such, each model element can be the target of only one containment reference. The two references in `BinaryExpression` mentioned earlier are both containment references, but for example the `clock` reference in `ClockReference` is not. Another special kind of reference are `inheritance` references to superclasses, indicated by large, white arrowheads. A class inherits all properties from its superclass, and elements of a class can be used instead of elements of the superclass whenever desired. The figure shows how the `Expression` class forms the basis of the package, as a superclass of nearly all other classes. In combination with the containment references, this can be used to create compound expressions, like `BinaryExpression` and `ListExpression`. This way, graphs of elements can be constructed that represent all desired CIF expressions.

Additionally, the metamodel contains three enumerations that describe what binary and unary operators and standard library functions exist. Note that there is no information given about these beyond their name, their semantics are defined elsewhere.

The UCIF expressions package is nearly identical to the CIF version. The main difference is that all class with names ending in "reference" are not present in UCIF. This is because the tools that create UCIF models cannot create the references that are required for these classes. Instead, there is a new class called `ReferenceExpression` that has one attribute called `name` that contains the text of the reference. Using this information, the references need for the desired CIF model can be reconstructed.

### 2.1.2 UCIF & CIF Types Metamodel

The `types` package of the CIF metamodel, shown if Figure 4, contains all elements that represent types. These elements are used both to represent declared types for declarations and types assigned to expressions. As in the `expressions` package, there is one main class, called `StaticType` in this case, that is a supertype of most other classes. And as in the `expressions` package, this is used to build compound types, like `DictionaryType` and `SetType`, from basic types.

The UCIF types package is identical to the CIF version, except that it is obviously not used to represent types assigned to expressions, but only to allow types to be defined for declarations.

### 3 MSOS

As mentioned before, type systems are a form of static semantics, which in turn is a part of the larger field of semantics. When semantics is given in the form of a mathematical model that describes the meaning of a program, it is referred to as formal semantics. The definition of formal semantics is a well-developed field, and various approaches to defining the semantics of languages exist. In these approaches three main classes can be discerned [23]: axiomatic semantics, denotational semantics and operational semantics. Axiomatic semantics [16] define meaning of statements by giving logical formulas that apply to those statements. An example of an axiomatic semantics formalism is Hoare logic [14]. Hoare logic uses the concept of state to define semantics. A state is described by a number of logical formulas, called `axioms`, and the meaning of a model element is expressed by the way it influences the state, as defined by
Figure 3: CIF expressions metamodel
Figure 4: CIF types metamodel
so-called *preconditions*, which describe the required state before the statement can execute, and *postconditions*, which describe the resulting state after the statement has been executed. For example, if we consider the semantics of the assignment statement, the preconditions could be used to guarantee that the expression providing the value can be computed, while the postconditions describe how the state changes to account for the new value of the variable being assigned. Axiomatic semantics formalisms are the most abstract of the semantics formalisms [23, p. 33], which makes them attractive for proving properties of the semantics they define.

Denotational semantics [27] attach meaning to models and model elements by, essentially, translating them. A *denotation* is a mathematical object that describes the meaning of language elements by relating them to a mathematical object called a *domain* that represents some generic semantics that are well-understood. A commonly used domain is that of partial functions. By using the right domain, one can use denotational semantics to provide intuitive descriptions of semantics.

The third main form of static semantics formalism is operational semantics [34, 36]. Operational semantics formalisms describe the meaning of constructs by describing an *abstract machine* that can execute them. The meaning of a program then becomes the result of executing the program by the machine, given the appropriate input. Initially, operational semantics were considered not abstract enough and inelegant [23, 26], but the development of methods like Structured Operational Semantics (SOS) by Plotkin [24] showed these were not fundamental issues, and that operational semantics could reach similar levels of mathematical expressiveness as denotational and axiomatic semantics [23]. SOS uses so-called rules that define a virtual machine by defining transitions between states. As shown in the SOS rule in Figure 3, the rule has an input element and environment on one side of the transition arrow and an output element and environment on the other side of the arrow. The input side is traditionally referred to as *left* and the output side as *right*, even if, like in this figure, the two sides are actually above and below the arrow. There are also a number of preconditions, shown above the line, that have to hold before the rule can be applied. In this example, the preconditions include a predicate on the input (the top precondition), a requirement that a certain step can be made (the middle precondition) and another predicate that updates an environment component (the bottom element). When used to define type systems, a state consists of elements of the model and the so-called *environment*. The environment is represented in the SOS rule in Figure 3 as a tuple with 9 fields that is combined with the input and output element of the rule into another tuple. The environment contains information about the context the rule is applied in that can be used in the rules.

As SOS specifications grow, there tend to be more rules that require the knowledge of specific context information, such as lists of declared variables and constants and their values [19]. SOS requires the component structure of the environment to be indicated on each and every rule. Hence, as the number of environment components grows, all rules become more complex, although some of the rules do not need the knowledge of the environment at all. Additionally, if the environment contains components that are not mentioned by a rule, that rule cannot be applied, which makes it more difficult to combine or modify SOS specifications, because the designer has to keep checking if all rules stay compatible with each other. Modular Structural Operational Semantics [20] (MSOS) were developed by Peter Mosses in an attempt to simplify the notation of SOS and to reduce the coupling between rules.

Similarly to SOS, a semantics definition in MSOS consists of a number of *rules*. An example MSOS rule is shown in Figure 3, several example rules can be found in Figure 10 in Section 5. Each rule describes all or part of the behavior of a specific language element. This is done via a transition from the construct before execution, described by an input pattern, to after execution, described by a target pattern. As a side effect, the environment can be updated. Besides an input pattern, rules can, and usually do, have preconditions that limit where the rules can be applied.
SOS rule

\[
\text{length}(element^*) > 0
\]
\[
< \text{element}^*, < Cl, Fi, F, GC, LC, P, S, T, V >>
\]
\[\Rightarrow\]
\[
< \text{element}_1^*, < Cl_1, Fi_1, F_1, GC_1, LC_1, P_1, S_1, T_1, V_1 >>
\]
\[
lubupdate(element_1^*.\text{type}, T_1) = t_e, T_2
\]

MSOS rule

\[
\text{length}(element^*) > 0
\]
\[
element^* \xrightarrow{T,X} element_1^*
\]
\[
lubupdate(element_1^*.\text{type}, T) = t_e, T_1
\]

Figure 5: Comparison between SOS and MSOS

MSOS removes the problem of unwieldy environments by storing environment information using transition labels. Figure 3 shows a comparison of two rules, one in SOS and one in MSOS form, that represent the same behavior. The exact details of the rules will be discussed in Section 5, but it is clear that the MSOS rule is less verbose than the SOS rule. The main reason for this size difference is that in MSOS, we do not have to explicitly mention environment components that are not mentioned in the current rule, but can let them be covered by the general X.

We choose MSOS over other formalisms because of the structure of the rules, with clear conditions leading to conclusions, and the modular nature of the formalism, opening the possibility of reusing type rules between different versions of the same language or even between languages.

### 3.1 Constraints

During the typing process, it can occur that we encounter an element that could potentially have one of several types assigned to it, without sufficient knowledge to choose between them at this point. However, as we encounter further elements, we may discover further knowledge that can help us reach a decision. The three main examples where this occurs are:

- **Number** expression elements can have one of three types: NatType, IntType and RealType.

- **ListExpression** and **SetExpression** elements are always of ListType and SetType, but also need a type for their elements. Normally, the type of the elements would be based on the types of the subexpressions providing the values for the elements, but if there are no subexpressions, because the list or set is empty, the type system still have to choose some type to complete the ListType or SetType.
- ReferenceExpressions that target overloaded functions from the standard library can refer to any of the implementations, that each have different types.

In each of these three cases, the type system cannot choose the correct type by analyzing expressions independently. The choice is influenced by elements that have been analyzed earlier or will be analyzed later. For example, if we consider a SetExpression with no elements, it might be later used in an assignment to a variable. The type system can then use the type of the variable to choose the type of the SetExpression. In another case, a SetExpression may be used as an argument of a function call. In that case, the type can be chosen based on type of the appropriate parameter of the type of the function.

In short, there are situations where a type checker needs information that is not available when it first encounters an element. One solution to this problem could be to assign no type to the element, and come back to it later when more information is available. When we tried to define the CIF type system using this idea, we found the type system became very verbose and not easy to understand. Instead, we choose to base the resolution of these issues on a paradigm known as constraint programming. Constraint programming is based on the notions of constraint variables and constraints. Constraint variables are used to represent points where choices have to be made, and constraints represent knowledge about those variables. For example, when we encounter a Number element, we introduce a constraint variable to represent the choice of types that has to be made there. Additionally, we introduce a constraint to represent the knowledge we already have about the variable: that its value must be one of three numeric types. Elsewhere in the model, we might encounter an element representing a comparison operation. There, a constraint can be introduced that state that the types of both subexpressions must be equal, even if it is not known at that point what the type actually is.

The process of finding suitable values for all constraint variables is called constraint solving. It is usually carried out by specialized pieces of software called constraint solvers. Constraint solvers can efficiently generate all combinations of values for constraint variables that satisfy all constraints. If there is no solution, we know the model contains errors that make it untypable. If there are solutions, an optimal solution from this set is selected, which in this case gives type values to all constraint variables. In essence, the constraint solver takes all type knowledge that has been collected in the constraints and combines it to choose a type for each element. In order to do this, the constraint solver does need to support values that are sufficiently complex and structured to describe all possible CIF types. As described in the types package of the metamodel, this includes containers of all sizes and with all kinds of elements and functions with all possible types of parameters. In practice, this means the domain must support values that have a name and a list of features, that can contain arbitrary nested values. How the constraint solver efficiently finds solutions for such domains is beyond the scope of this document. An example of a constraint engine that can handle the types of constraints that are used here is ECLiPSe [21]. A discussion of related work in the field of constraints-based typing can be found in Section 8.

4 Preliminaries

Because MSOS is a generic semantics formalism, it does not address CIF-specific functionality directly. For example, preconditions are pure logical formulas, with no intrinsic distinction between elements, properties of elements and other values. We use a number of notation conventions to make these distinctions clearer, as described in Section 4.1. Another example of a generic feature that is used in a CIF-specific way is the environment. In principle, MSOS environments can consist of arbitrary components, but the CIF specification uses a specific set.
This set is introduced in Section 4.2. Finally, there are certain concepts recurring throughout
the specification. For example, knowledge about how CIF types relate to each other is used in
all MSOS rules where multiple types are possible. In order to make the rules more compact
and easier to read, we have created a number of functions that express these recurring concepts.
These are defined and explained in Section 4.4.

4.1 Typography

In this document we use the formatting of names to indicate their purpose. The following styles
are used:

- functions
- type and enumeration values
- properties
- variables

In addition, all names that end with a superscript asterisk are lists. Note that \( t \) and \( t^* \)
are separate variables, and \( t^* \) does not refer to a list of copies of \( t \). If we want to refer to
specific elements in a list, we use \( t_i \) for the \( i \)th element. Lists can be concatenated using `++`
Conceptually, all lists are constructed from two constructors: \( \varepsilon \) for empty lists and `:` to add an
element to the front of a list.

4.2 Environment

Some of the constructs that form part of the CIF language are references to other elements, called
declarations, in the model, as is shown in Figure 3. In UCIF, as described in Section 2.1.1, these
references are not represented explicitly, but implicitly in the form of \( \text{ReferenceExpression} \)
elements that initially contain only a name. In a correctly typed model, there is only one correct
target declaration for each such expression. Which declaration this is, depends on the name
given in the \( \text{ReferenceExpression} \). The process of determining what can be referenced by a
given element in the model is called scoping. For example, a \( \text{ReferenceExpression} \) containing
the name “volume” can only refer to elements named “volume”. Additionally, the element to be
referenced should be of an appropriate type. In CIF, only declarations can be referenced, two
\( \text{ReferenceExpression} \) cannot refer to each other. Finally, the structure of the model determines
what can be referenced. For example, in CIF, a so-called \( \text{Scope} \) can contain a number of decla-
rations that can only be referenced by elements contained in the \( \text{Scope} \). References outside the
\( \text{Scope} \) can refer to elements with the same name not contained in the \( \text{Scope} \), allowing multiple
elements with the same name to exist in one CIF specification. For example, a CIF specification
can contain multiple \( \text{Scope} \) elements, each containing a \( \text{Constant} \) named “x”, without interfering
with each other.

Scoping is closely related to the type system, because the potential target of a reference can
depend on types of other elements. In CIF, this occurs when dealing with tuples. Tuples are data
structures that contain named fields. If we have a variable that contains a value of a tuple type,
fields in it can be accessed. However, we have to know what the type of the variable is before we
know what fields can be accessed, and what types the fields have. For example, let \( \text{exampleA} \) be
a tuple type with fields \( a \) and \( b \), and let \( \text{exampleB} \) be a tuple type with fields \( c \) and \( d \). We may
then want to access field \( a \) of a certain tuple. We can do this by using a so-called \( \text{projection} \),
which is applied to a tuple instance and a reference to field \( a \). However, before typing, we do not
know whether the tuple actually is of type \texttt{exampleA}. Examples where this can occur are tuple expressions, where tuples are defined directly, or by overloaded functions that return different tuples depending on the type of the arguments. If the tuple is actually of type \texttt{exampleB}, it has no field \texttt{a} that can be referenced. Thus, we can only decide whether a reference to a tuple field is valid after we have determined the type of the tuple that the actual data will come from.

Nevertheless, we have tried to separate the scoping issue from the type system, in order to reduce the complexity of the type system. The information computed during scoping is represented by collecting all possible targets of references. In MSOS, the label of the transition is used to store this environment information. Information that cannot be discovered during the scoping phase, like in the tuple example mentioned above, is added to the environment when we discover it.

The environment as used in this document consists of several components, each containing instances of a specific type of target. We choose to use this environment configuration because the CIF metamodel requires that each reference is classified according to the type of the referenced declaration. By splitting the environment, we can easily determine the type of the target by considering the component it was retrieved from. When using MSOS, it is customary to only mention the components of the environment that are needed for the current rule. The other components are collectively referred to as \textit{X}. For example, in the rule for \texttt{TypeVariable} elements, the \texttt{typeenv} components of the environment is accessed as follows: \texttt{typeenv = TENV,X}. This means that the \texttt{typeenv} component is stored in \texttt{TENV}, so elements can be retrieved from or added to it, while the rest of the environment remains anonymous. The components currently used are:

\texttt{constraintenv} contains the constraints generated during the typing process.

\texttt{clockenv} relates clock names to their declarations, i.e. this component contains declarations of type \texttt{Clock}.

\texttt{fieldenv} relates field names to their declarations. Fields are declared as part of tuple types. This component is usually empty, except in the context of a tuple-by-name projection. It contains declarations of type \texttt{Field}.

\texttt{functionenv} relates user-defined function names to their declarations. Overloading is not allowed for these functions, so names must be unique. This component contains declarations of type \texttt{InternalFunctionDeclaration}.

\texttt{gconstenv} relates global constant names to their declarations. Global constants are constants that are defined for entire specifications, and apply to all models, functions, automata and other declarations that have been defined as part of the specification. This component contains declarations of type \texttt{ConstantDeclaration}.

\texttt{lconstenv} relates local constant names to their declarations. Local constants are defined for a part of a specification, like a model, and cannot be accessed outside of that part. This component contains declarations of type \texttt{Constant}.

\texttt{parameterenv} relates parameter names to their declarations. Parameters are declared as part of functions. This component contains declarations of type \texttt{Parameter}.

\texttt{stdlibenv} relates standard-library function names to their declarations. The list of functions contained in the standard library is defined as part of the metamodel. Example standard library functions are trigonometric functions like \texttt{sin}, \texttt{cos} and \texttt{tan}, that compute sine, cosine and tangent values respectively. Since overloading is allowed these functions, names
of elements in this component need not be unique. This component contains declarations of type `ExternalFunctionDeclaration`.

`typeenv` relates type names to their declarations. This component contains declarations of type `TypeDeclaration`, and constraints created by typing rules.

`varenv` relates variable names to their declarations. This component contains declarations of type `Variable`.

### 4.3 Element notation

In this document, we often have to refer to elements of the UCIF input model and the CIF output model. As described in Section 2.1, all elements in both input and output models are described by the corresponding metamodels. We can use this to define patterns for model elements, consisting of at least the name of the desired EClass and if necessary a number of expressions referring to its properties. An example of such a pattern is given in Figure 6. In this case, the pattern describes elements that conform to the EClass `BinaryExpression`, with properties `operator`, `leftChild` and `rightChild` represented by variables `o`, `lhs` and `rhs` respectively. If used as an input pattern for a rule, this means we essentially read the values of those properties into the variables. If used as an output pattern of a rule, the properties of the new model element get their values from the variables. Both for input patterns and output patterns, note that elements matched may contain more properties than those mentioned in the pattern, but that are not considered relevant in the case of this rule. For example, it is possible to store information about position in a source document in the element, but in the case of CIF this does not influence the type system. In order to simplify the rules, decrease their size, and to make them more generic, we avoid mentioning unnecessary properties. Any properties present in both the input and output element but not mentioned in the rules are assumed to be copied implicitly.

### 4.4 Functions

This section describes the auxiliary functions that are used in the rules in this document. Functions add CIF-specific functionality that does not exist in MSOS in its generic form. The functions can be divided into two groups. The first group deals with the creation of the constraints that are used to compute the values of type variables that are created during the typing process. The second group is used to manipulate data structures that are used in this document, like sets and dictionaries, to store information while maintaining certain properties. For example, sets are used whenever duplicates are unnecessary or undesired. In order to use these data structures, we need to be able to read and update them, but full details of their implementation lie beyond the scope of this document.

The functions described here all have several parameters as part of their signature. In order to define a function, we consider one or more possible values for the parameters, and then give the result of the function when invoked with those parameter values. As example, if we consider the `isSubType` function in the next section, the first line of the definition describes the case where both parameters are equal, as represented by the variable `T`. In that case, the function returns true. The second line describes the case where the first parameter has a specific value, but the second parameter can have any value. The third line shows a case where the values of both parameters are set. A more complex case, like in line six, can involve boolean expressions using the variables, that describe how the result of the function depends on the parameters.
4.4.1 isSubType

This function describes the subtype relation between types. In this case, there can be multiple applicable rules for a single call, but the general intention is that $S$ is a subtype of $T$ if at least one rule applies, otherwise, the result is false. The values ending in Type used here refer to elements of the types subpackage of the CIF metamodel. Note that isSubType only handles actual type values, not type variables.

1. isSubType($T, T$) = true
2. isSubType(VoidType, $T$) = true
3. isSubType(NatType, IntType) = true
4. isSubType(NatType, RealType) = true
5. isSubType(IntType, RealType) = true
6. isSubType(ListType(elementType = $t_1$), ListType(elementType = $t_2$)) = isSubType($t_1, t_2$)
7. isSubType(SetType(elementType = $t_1$), SetType(elementType = $t_2$)) = isSubType($t_1, t_2$)
8. isSubType(ArrayType(elementType = $t_1$, dimension = $n$), ArrayType(elementType = $t_2$, dimension = $n$)) = isSubType($t_1, t_2$)
9. isSubType(VectorType(elementType = $t_1$, dimension = $n$), ArrayType(elementType = $t_2$, dimension = $n$)) = isSubType($t_1, t_2$)
10. isSubType(MatrixType(elementType = $t_1$, rows = $m$, columns = $n$), MatrixType(elementType = $t_2$, rows = $m$, columns = $n$)) = isSubType($t_1, t_2$)
11. isSubType(DictionaryType(key = $t_{k1}$, value = $t_{v1}$), DictionaryType(key = $t_{k2}$, value = $t_{v2}$)) = isSubType($t_{k1}, t_{k2}$) ∧ isSubType($t_{v1}, t_{v2}$)
12. isSubType(TupleType(fields = $f^*$), TupleType(fields = $g^*$)) = ($\forall i :$ isSubType($f_i.type, g_i.type$))

4.4.2 lubupdate

The least upper bound update function, or lubupdate for short, takes a list of types and a type environment as parameters, and computes a least upper bound of the list of types, using the relation defined by the isSubType function as a partial order. In other words, it returns a
pair consisting of the smallest type that is a supertype of all type arguments, according to the isSubType function. Additionally an updated version of the type environment is returned, that contains any new constraints found.

1. \[
\text{\texttt{lubupdate}}(S : T, E) = U, E \text{ if } \text{isSubType}(S, U) \land \text{isSubType}(T, U) \land \\
(\forall V : (\text{isSubType}(S, V) \land \text{isSubType}(T, V)) \Rightarrow \text{isSubType}(U, V))
\]

2. \[
\text{\texttt{lubupdate}}\left(\text{\texttt{TypeVariable}}\left(\begin{array}{l}
\text{name } = n
\end{array}\right) : T, E\right) = T, \text{addConstraint}(n = T, E)
\]

3. \[
\text{\texttt{lubupdate}}\left(S : \text{\texttt{TypeVariable}}\left(\begin{array}{l}
\text{name } = n
\end{array}\right), E\right) = S, \text{addConstraint}(n = S, E)
\]

4. \[
\text{\texttt{lubupdate}}\left(\text{\texttt{TypeVariable}}\left(\begin{array}{l}
\text{name } = n_s
\end{array}\right) : \text{\texttt{TypeVariable}}\left(\begin{array}{l}
\text{name } = n_t
\end{array}\right), E\right) = \text{\texttt{TypeVariable}}\left(\begin{array}{l}
\text{name } = n
\end{array}\right), \text{addConstraint}(n = n_t, \text{addConstraint}(n = n_s, E, E))
\]

5. \[
\text{\texttt{lubupdate}}(S_1 : S^*, E) = T_2, E_2 \text{ if } \text{\texttt{lubupdate}}(S^*, E) = T_1, E_1 \land \text{\texttt{lubupdate}}(S_1 : T_1, E_1) = T_2, E_2
\]

6. \[
\text{\texttt{lubupdate}}(S : \varepsilon, E) = S, E
\]

4.4.3 addConstraint

addConstraint\((cs, C)\) adds new constraints \(cs\) to the constraint store \(C\) from the environment. If this results in a situation where there are no more possible solutions, the condition is considered to have failed.

4.4.4 addParameterConstraints

addParameterConstraints\((p^*, a^*, E)\) adds a set of constraints for the arguments \(a^*\) of a function call to match them to the parameters \(p^*\) of the function to be invoked in environment \(E\). For the function call to be valid, \(a^*\) and \(p^*\) should be of equal length. We then know that argument \(a_i\) should produce a value for parameter \(p_i\), and we can create a constraint that expresses that the types should be compatible.

4.4.5 makeField

This function creates fields to be used to construct a TupleType, for example when typing a TupleExpression. All fields have types, but not all fields have a explicit name. Therefore, this function takes one or two lists as arguments, an optional list of names and a list of types. The function creates a Field object for each type.

1. \[
\text{\texttt{makeField}}(T_1) = \text{\texttt{Field}}(\text{\texttt{name} } = \varepsilon, \text{\texttt{type} } = T_1)
\]

2. \[
\text{\texttt{makeField}}(T_1 : T^*) = \text{\texttt{makeField}}(T_1) : \text{\texttt{makeField}}(T^*)
\]

3. \[
\text{\texttt{makeField}}(N_1, T_1) = \text{\texttt{Field}}(\text{\texttt{name} } = N_1, \text{\texttt{type} } = T_1)
\]
4. makeField\((N_1 : N^*, T_1 : T^*) = \texttt{Field}(name = N_1, type = T_1) : \texttt{makeField}(N^*, T^*)\)

4.4.6 mergeFields

This function resolves name conflicts when merging two lists of fields, for example when two
tuples are concatenated together. All fields of the second that do not have the same name as
a field in the first list are added to the first list. All names are then stripped from the list
by invoking the makeField to create a new field with only a type, to create a combined list of
unnamed fields.

1. mergeFields\((F_1 : F^*, \varepsilon) = \texttt{makeField}(F_1, \texttt{type}) : \texttt{mergeFields}(F^*, \varepsilon)\)

2. mergeFields\((F^*, G_1 : G^*) = \texttt{mergeFields}(F^*, G^*) \text{ if } \exists F_1: F_1 \cdot \texttt{name} = G_1\cdot \texttt{name}\)

3. mergeFields\((F^*, G_1 : G^*) = \texttt{mergeFields}(F^* \cdot G_1 : \varepsilon, G^*) \text{ if } \neg \exists F_1 : F_1 \cdot \texttt{name} = G_1 \cdot \texttt{name}\)

4.4.7 eval

eval\(e\) returns the value of expression \(e\) if it can be determined at compile-time. The details
of what can be determined depend on the compiler, but in general all values used must be
compile-time constants.

4.4.8 any

This functions returns a random member of a given set.

4.4.9 update

update\((n, e, d)\) returns a new dictionary \(d'\) with the value for key \(n\) replaced with \(e\) if \(n\) is a
key of dictionary \(d\), or a new dictionary \(d'\) with new key \(n\) with value \(e\) if \(n\) is not a key of \(d\).

4.4.10 lookUp

lookUp\((n, d)\) returns one of the values corresponding to \(n\) if \(n\) is a key of a dictionary \(d\). If there
are multiple values, the choice is considered random.

4.4.11 lookUpSet

lookUpSet\((n, d)\) returns all of the values corresponding to \(n\) if \(n\) is a key of dictionary \(d\).

4.4.12 length

length\((t^*)\) returns the length of list \(t^*\)

4.4.13 getDistributionTypes

getDistributionTypes\((n)\) return the set of distributions defined for the name \(n\). This information
comes from the predefined table shown below. Each distribution has one or more parameters
and returns one value of the given type. Note that the seed values are always of NatType, so
that is not repeated below.
### Distribution Parameters Result

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Parameters</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>BoolType</td>
<td>BoolType</td>
</tr>
<tr>
<td>Constant</td>
<td>NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Constant</td>
<td>IntType</td>
<td>IntType</td>
</tr>
<tr>
<td>Constant</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Bernoulli</td>
<td>RealType</td>
<td>BoolType</td>
</tr>
<tr>
<td>Bernoulli</td>
<td>RealType</td>
<td>NatType</td>
</tr>
<tr>
<td>Beta</td>
<td>RealType, RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Binomial</td>
<td>RealType, NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Erlang</td>
<td>NatType, RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Exponential</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Gamma</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Geometric</td>
<td>RealType</td>
<td>NatType</td>
</tr>
<tr>
<td>LogNormal</td>
<td>RealType, RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Normal</td>
<td>RealType, RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Poisson</td>
<td>RealType</td>
<td>NatType</td>
</tr>
<tr>
<td>Random</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Triangle</td>
<td>RealType, RealType, RealType</td>
<td>NatType</td>
</tr>
<tr>
<td>Uniform</td>
<td>NatType, NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Uniform</td>
<td>IntType, IntType</td>
<td>IntType</td>
</tr>
<tr>
<td>Uniform</td>
<td>RealType, RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Weibull</td>
<td>RealType, RealType</td>
<td>RealType</td>
</tr>
</tbody>
</table>

#### 4.4.14 getOpTypes

getOpTypes(n) returns the set of types for the available operations with name n. This information comes from the predefined table shown below. The table mostly consist of binary operators, with two unary exceptions, Inverse and Negate. Only operations on basic types are listed in the table, operations on compound types are treated separately.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Left</th>
<th>Right</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>NatType</td>
<td>NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Addition</td>
<td>IntType</td>
<td>IntType</td>
<td>IntType</td>
</tr>
<tr>
<td>Addition</td>
<td>RealType</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>ConditionalConjunction</td>
<td>BoolType</td>
<td>BoolType</td>
<td>BoolType</td>
</tr>
<tr>
<td>ConditionalDisjunction</td>
<td>BoolType</td>
<td>BoolType</td>
<td>BoolType</td>
</tr>
<tr>
<td>Conjunction</td>
<td>BoolType</td>
<td>BoolType</td>
<td>BoolType</td>
</tr>
<tr>
<td>Disjunction</td>
<td>BoolType</td>
<td>BoolType</td>
<td>BoolType</td>
</tr>
<tr>
<td>Division</td>
<td>NatType</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Division</td>
<td>IntType</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Division</td>
<td>RealType</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>FloorDivision</td>
<td>NatType</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>FloorDivision</td>
<td>IntType</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>GreaterEqual</td>
<td>NatType</td>
<td>NatType</td>
<td>BoolType</td>
</tr>
<tr>
<td>GreaterEqual</td>
<td>IntType</td>
<td>IntType</td>
<td>BoolType</td>
</tr>
<tr>
<td>GreaterEqual</td>
<td>RealType</td>
<td>RealType</td>
<td>BoolType</td>
</tr>
<tr>
<td>GreaterEqual</td>
<td>StringType</td>
<td>StringType</td>
<td>BoolType</td>
</tr>
<tr>
<td>GreaterThan</td>
<td>NatType</td>
<td>NatType</td>
<td>BoolType</td>
</tr>
<tr>
<td>Operation</td>
<td>Left</td>
<td>Right</td>
<td>Result</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------</td>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>GreaterThan</td>
<td>IntType</td>
<td>IntType</td>
<td>BoolType</td>
</tr>
<tr>
<td>GreaterThan</td>
<td>RealType</td>
<td>RealType</td>
<td>BoolType</td>
</tr>
<tr>
<td>GreaterThan</td>
<td>StringType</td>
<td>StringType</td>
<td>BoolType</td>
</tr>
<tr>
<td>Implication</td>
<td>BoolType</td>
<td>BoolType</td>
<td>BoolType</td>
</tr>
<tr>
<td>Inverse</td>
<td>NatType</td>
<td></td>
<td>IntType</td>
</tr>
<tr>
<td>Inverse</td>
<td>IntType</td>
<td></td>
<td>IntType</td>
</tr>
<tr>
<td>Inverse</td>
<td>RealType</td>
<td></td>
<td>RealType</td>
</tr>
<tr>
<td>LessEqual</td>
<td>NatType</td>
<td>NatType</td>
<td>BoolType</td>
</tr>
<tr>
<td>LessEqual</td>
<td>IntType</td>
<td>IntType</td>
<td>BoolType</td>
</tr>
<tr>
<td>LessEqual</td>
<td>RealType</td>
<td>RealType</td>
<td>BoolType</td>
</tr>
<tr>
<td>LessEqual</td>
<td>StringType</td>
<td>StringType</td>
<td>BoolType</td>
</tr>
<tr>
<td>LessThan</td>
<td>NatType</td>
<td>NatType</td>
<td>BoolType</td>
</tr>
<tr>
<td>LessThan</td>
<td>IntType</td>
<td>IntType</td>
<td>BoolType</td>
</tr>
<tr>
<td>LessThan</td>
<td>RealType</td>
<td>RealType</td>
<td>BoolType</td>
</tr>
<tr>
<td>LessThan</td>
<td>StringType</td>
<td>StringType</td>
<td>BoolType</td>
</tr>
<tr>
<td>Maximum</td>
<td>NatType</td>
<td>NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Maximum</td>
<td>IntType</td>
<td>IntType</td>
<td>NatType</td>
</tr>
<tr>
<td>Maximum</td>
<td>NatType</td>
<td>IntType</td>
<td>NatType</td>
</tr>
<tr>
<td>Maximum</td>
<td>IntType</td>
<td>IntType</td>
<td>NatType</td>
</tr>
<tr>
<td>Maximum</td>
<td>RealType</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Maximum</td>
<td>StringType</td>
<td>StringType</td>
<td>StringTy</td>
</tr>
<tr>
<td>Minimum</td>
<td>NatType</td>
<td>NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Minimum</td>
<td>IntType</td>
<td>IntType</td>
<td>IntType</td>
</tr>
<tr>
<td>Minimum</td>
<td>RealType</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Minimum</td>
<td>StringType</td>
<td>StringType</td>
<td>StringTy</td>
</tr>
<tr>
<td>Modulus</td>
<td>NatType</td>
<td>NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Modulus</td>
<td>IntType</td>
<td>IntType</td>
<td>IntType</td>
</tr>
<tr>
<td>Multiplication</td>
<td>NatType</td>
<td>NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Multiplication</td>
<td>IntType</td>
<td>IntType</td>
<td>IntType</td>
</tr>
<tr>
<td>Multiplication</td>
<td>RealType</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Negate</td>
<td>BoolType</td>
<td></td>
<td>BoolType</td>
</tr>
<tr>
<td>Power</td>
<td>NatType</td>
<td>NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Power</td>
<td>IntType</td>
<td>IntType</td>
<td>IntType</td>
</tr>
<tr>
<td>Power</td>
<td>IntType</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Power</td>
<td>RealType</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Subtraction</td>
<td>NatType</td>
<td>NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Subtraction</td>
<td>IntType</td>
<td>IntType</td>
<td>IntType</td>
</tr>
<tr>
<td>Subtraction</td>
<td>RealType</td>
<td>RealType</td>
<td>RealType</td>
</tr>
</tbody>
</table>

### 4.4.15 getStdLibTypes

getStdLibTypes(n) returns the set of functions defined in the standard library for name n.

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs</td>
<td>IntType</td>
<td>NatType</td>
</tr>
<tr>
<td>Abs</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Acos</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Acos</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Function</td>
<td>Parameters</td>
<td>Result</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td>Acos</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Acosh</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Acosh</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Acosh</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Asin</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Asin</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Asin</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Asinh</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Asinh</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Asinh</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Atan</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Atan</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Atan</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Atanh</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Atanh</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Atanh</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Bool2String</td>
<td>BoolType</td>
<td>StringType</td>
</tr>
<tr>
<td>Cbrt</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Cbrt</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Cbrt</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Ceil</td>
<td>NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Ceil</td>
<td>IntType</td>
<td>IntType</td>
</tr>
<tr>
<td>Ceil</td>
<td>RealType</td>
<td>IntType</td>
</tr>
<tr>
<td>Cos</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Cos</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Cos</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Cosh</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Cosh</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Cosh</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Drop</td>
<td>ListType</td>
<td>ListType</td>
</tr>
<tr>
<td>Drop</td>
<td>StringType</td>
<td>StringType</td>
</tr>
<tr>
<td>Exp</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Exp</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Exp</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Floor</td>
<td>RealType</td>
<td>IntType</td>
</tr>
<tr>
<td>Head</td>
<td>ListType</td>
<td>t_e</td>
</tr>
<tr>
<td>HeadReverse</td>
<td>ListType</td>
<td>t_e</td>
</tr>
<tr>
<td>Insert</td>
<td>ListType</td>
<td>ListType</td>
</tr>
<tr>
<td>Int2Nat</td>
<td>IntType</td>
<td>NatType</td>
</tr>
<tr>
<td>Function</td>
<td>Parameters</td>
<td>Result</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Int2Real</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Int2String</td>
<td>IntType</td>
<td>StringType</td>
</tr>
<tr>
<td>Length</td>
<td></td>
<td>NatType</td>
</tr>
<tr>
<td></td>
<td>ListType (elementType = te)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetType (elementType = te)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>StringType</td>
<td></td>
</tr>
<tr>
<td>Ln</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Ln</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Ln</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Log</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Log</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Log</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Nat2Int</td>
<td>NatType</td>
<td>NatType</td>
</tr>
<tr>
<td>Nat2Real</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Nat2String</td>
<td>NatType</td>
<td>StringType</td>
</tr>
<tr>
<td>Real2String</td>
<td>RealType</td>
<td>StringType</td>
</tr>
<tr>
<td>Round</td>
<td>RealType</td>
<td>IntType</td>
</tr>
<tr>
<td>SetSeed</td>
<td>DistributionType (resultType = te), NatType</td>
<td>DistributionType (resultType = te)</td>
</tr>
<tr>
<td>Sign</td>
<td>IntType</td>
<td>IntType</td>
</tr>
<tr>
<td>Sign</td>
<td>RealType</td>
<td>IntType</td>
</tr>
<tr>
<td>Sin</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Sin</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Sin</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Sinh</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Sinh</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Sinh</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Sqrt</td>
<td>NatType</td>
<td>RealType</td>
</tr>
<tr>
<td>Sqrt</td>
<td>IntType</td>
<td>RealType</td>
</tr>
<tr>
<td>Sqrt</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>String2Bool</td>
<td>StringType</td>
<td>BoolType</td>
</tr>
<tr>
<td>String2Int</td>
<td>StringType</td>
<td>IntType</td>
</tr>
<tr>
<td>String2Nat</td>
<td>StringType</td>
<td>NatType</td>
</tr>
<tr>
<td>String2Real</td>
<td>StringType</td>
<td>RealType</td>
</tr>
<tr>
<td>Tail</td>
<td>ListType (elementType = te)</td>
<td>te</td>
</tr>
<tr>
<td>TailReverse</td>
<td>ListType (elementType = te)</td>
<td>ListType (elementType = te)</td>
</tr>
<tr>
<td>Take</td>
<td>ListType (elementType = te), NatType</td>
<td>ListType (elementType = te)</td>
</tr>
<tr>
<td>Take</td>
<td>StringType, NatType</td>
<td>StringType</td>
</tr>
<tr>
<td>Tan</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Tanh</td>
<td>RealType</td>
<td>RealType</td>
</tr>
<tr>
<td>Function</td>
<td>Parameters</td>
<td>Result</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td>Transpose</td>
<td>( \text{MatrixType} )</td>
<td></td>
</tr>
</tbody>
</table>
| \( \begin{align*} 
\text{columnDimension} &= n \\
\text{elementOperator} &= o \\
\text{elementType} &= t_e \\
\text{rowDimension} &= m 
\end{align*} \) | \( \text{MatrixType} \) |
| \( \begin{align*} 
\text{columnDimension} &= m \\
\text{elementOperator} &= o \\
\text{elementType} &= t_e \\
\text{rowDimension} &= n 
\end{align*} \) |

5 Example rules

In the next section, we describe a set of MSOS rules that together define the CIF type system. By applying these rules to expressions, like the example list expression shown in Figure 7, we can derive their type. The figure shows a list expression containing two numeric elements, \([1, 1.5]\) in model form, using the notation introduced in Section 4.3. When we want to compute a type for this expression, first we need to find the MSOS rule which applies to the outermost element. In order to find this rule, we first look for rules that have a \text{ListExpression} as their left-hand side or input pattern. There are two of such rules, as shown in Figure 10 in the section “List Expression”. In the Type system section, these rules can be found in Section 7.16.

![Figure 7: Example ListExpression element](image)

\[
\begin{align*}
\text{ListExpression} &\xrightarrow{X} \\
(\text{elements} &= \varepsilon) &\Rightarrow
(\text{elements} = \varepsilon \\
\text{type} &\Rightarrow \text{ListType(elementType} = \text{TypeVariable}) \\
&\text{name} = I)
\end{align*}
\]

Figure 8: Rule for empty list expression

The first rule, shown in Figure 8, is stated to handle empty lists. If we try to actually match our expression element to the input pattern we find it does not fit: the pattern requires the \text{elements} property of the \text{ListExpression} to be empty, while it contains two elements, a \text{Number} element and a \text{RealNumber} element. This means this rule will not produce any types, and we stop trying to apply it and instead try to apply the next rule, shown in Figure 9.

\[
\begin{align*}
\text{length(element*)} &> 0 \\
\text{element* } &\xrightarrow{T} \text{element1*} \\
\text{lubupdate(element1*.type}, T) &\Rightarrow t_e, T1
\end{align*}
\]

![Figure 9: Rule for non-empty list expressions](image)

\[
\begin{align*}
\text{ListExpression} &\xRightarrow{T, typeenv'} \\
(\text{elements} = \text{element*}) &\Rightarrow
(\text{elements} = \text{element1*} \\
\text{type} &\Rightarrow \text{ListType(elementType} = t_e)
\end{align*}
\]

If we match the \text{ListExpression} to the input pattern of this rule, we find it matches, with the value of the \text{elements} property being assigned to the variable \text{element*}. 

20
Constants

Number

\[ \text{addConstraint}(I \in \{\text{NatType}, \text{IntType}, \text{RealType}\}, T) = T_1 \]

\[
\begin{array}{c}
\text{Number} \\ ()
\end{array} \quad \text{typeenv} = T, \text{typeenv'} = T_1, X
\]

\[
\begin{array}{c}
\text{Number} \\ ( \text{type} = \text{TypeVariable} ) \\
\quad ( \text{name} = I )
\end{array}
\]

RealLiteral

\[
\begin{array}{c}
\text{RealLiteral} \\ ()
\end{array}
\]

\[
\begin{array}{c}
\text{RealNumber} \\
\quad ( \text{type} = \text{RealType} )
\end{array}
\]

StringLiteral

\[
\begin{array}{c}
\text{StringLiteral} \\ ()
\end{array}
\]

\[
\begin{array}{c}
\text{StringLiteral} \\
\quad ( \text{type} = \text{StringType} )
\end{array}
\]

ListExpression

Empty

\[
\begin{array}{c}
\text{ListExpression} \\
\quad ( \text{elements} = \varepsilon )
\end{array}
\]

\[
\begin{array}{c}
\text{ListExpression} \\
\quad ( \text{elements} = \varepsilon )
\end{array}
\]

\[
\begin{array}{c}
\text{ListExpression} \\
\quad ( \text{elements} = \text{element}^* )
\end{array}
\]

Nonempty

\[
\begin{array}{c}
\text{length}(\text{element}^*) > 0
\end{array}
\]

\[
\begin{array}{c}
\text{element}^* \xrightarrow{X} \text{element}^1
\end{array}
\]

\[
\begin{array}{c}
\text{lubupdate}(\text{element}^1.\text{type}, T) = t_e, T_1
\end{array}
\]

BinaryExpression

Equal & NotEqual

\[
\alpha \in \{\text{Equal, NotEqual}\}
\]

\[
\begin{array}{c}
\text{lhs} \xrightarrow{X} \text{lhs}'
\end{array}
\]

\[
\begin{array}{c}
\text{rhs} \xrightarrow{X} \text{rhs}'
\end{array}
\]

\[
\begin{array}{c}
\text{lubupdate} (\text{lhs}'.\text{type} : \text{rhs}'.\text{type}, T) = t, T_1
\end{array}
\]

Figure 10: Rules used in examples
The next step is to check whether the preconditions of this rule hold. In this case, there are three preconditions: \( \text{length}(\text{element}^*) > 0 \), \( \text{element}^* \xrightarrow{\Delta} \text{element1}^* \) and \( \text{lubupdate}(\text{element1}^*.\text{type}, T) = t_c, T_1 \). The first precondition checks if \( \text{element}^* \) is non-empty. This guarantees that the first rule and the current rule cannot be applied to the same element, because a list can never be empty and nonempty at the same time, thus eliminating any (potential) conflict. The next precondition requires that we can (recursively) construct typed versions of all model elements in \( \text{element}^* \). In the example case, we can use the first rule of Figure 10 for the \( \text{Number} \) element, and the second rule for the \( \text{RealNumber} \) element. In the latter case, there is only one possible result type, namely \( \text{RealType} \). In the former case, the result can be one of several numeric types. In such cases, we represent the type of the expression as a constraint variable, \( t_e \). During the typing process, we can further refine the constraints based on applied rules, as long as all constraints remain satisfiable. In the example, during the typing of both the \( \text{Number} \) and the \( \text{RealType} \), constraints are added that restrict the value of \( t_e \) to numeric types and to \( \text{RealType} \) respectively. In the end, we can choose any type that meets all constraints found as our final result for this expression.

The intermediate results are stored in a new list, \( \text{element1}^* \).

If all subexpressions are successfully typed, we check whether the intersection of the possible types of the subexpressions is not empty under the constraints that have been found. In our example, we have two constraints. The first constraint limits the first subexpression to types that can be widened to \( \text{RealType} \). The second constraint limits the second subexpression to \( \text{RealType} \) only. Because all subexpression can be assigned \( \text{RealType} \), we get a new type variable, \( t_e \), that represent the common type of all subexpressions. We use this type and the typed versions of the subexpressions to construct a typed version of the \( \text{ListExpression} \), as indicated by the right-hand side, or target, pattern. Note that we do not use \( t_e \) as the type for the result directly, but first construct a \( \text{ListType} \) object that uses \( t_e \) as its element type. The end result is that the list expression is determined to have type \( \text{ListType} \) with \( \text{elementType} \) depending on the types of the elements of the list. Additionally, the environment is updated with new constraints. In concordance with MSOS convention, \( \text{typeenv} \) refers to the value before the transition is executed, and \( \text{typeenv}' \) to the new value after the transition is executed.

\[
\begin{align*}
\text{ListExpression} & = \left( \begin{array}{l}
\text{Number} \\
\text{RealNumber} \\
\text{ListType}
\end{array} \right) \\
\text{elements} & = ( \begin{array}{l}
\text{value} = 1 \\
\text{type} = \text{RealType}
\end{array} ), ( \begin{array}{l}
\text{value} = 1.5 \\
\text{type} = \text{RealType}
\end{array} ) \\
\text{type} & = ( \text{elementType} = \text{RealType} )
\end{align*}
\]

Figure 11: Example typed \( \text{ListExpression} \) element

It is also possible that typing reaches a situation where no rule can be applied. Consider for example the expression shown in Figure 12. The example shows the model elements representing the expression “hello’=1.5”. In other words, a string is compared to a number. If we try to apply the rule for the \( \text{Equal} \) operator shown if Figure 7.16, we see it has 4 preconditions. The first precondition state the rule applies to the \( \text{Equal} \) and \( \text{NotEqual} \) operators, which is the case here. The next two preconditions state that it must be possible to type the expression in properties \( \text{lhs} \) and \( \text{rhs} \). In this case, those expression do indeed have valid types, \( \text{StringType} \) and \( \text{RealType} \). So far, there are no obstacles to applying the rule. However, according to the CIF semantics, only values of the same type can be compared. This is stated in the last precondition, that adds a constraint stating such to the constraint store. Because \( \text{StringType} \) and \( \text{RealType} \) are clearly not equal, this constraint can never be satisfied. Thus, this precondition is considered to have
failed, and the rule cannot be applied. Because there are no other rules that can be applied to this operation, this element cannot be typed and typing fails.

\[
\begin{align*}
\text{BinaryExpression} & = \begin{cases} 
\text{Equal} \\
\text{StringLiteral} & (\text{value} = \text{'hello'}) \\
\text{RealNumber} & (\text{value} = 1.5)
\end{cases}
\end{align*}
\]

Figure 12: Example BinaryExpression element

6 Type system

6.1 Type expressions

In this document, we describe a type system that defines the type of CIF expressions, i.e., elements of classes that are a subclass of Expression. Types are not technically part of this set, and according the metamodel, they do not have a type property. However, because types both contain expressions, and are used in the type system, we still give rules here that define what constitutes a proper type. Note that apart from any expressions occurring in them, the type elements do not change when they are typed.

6.1.1 ArrayType

Arrays contain a set number of elements of identical types. Thus, ArrayType has two properties, elementType, which has to contain a proper type, and dimension, which has to be a positive, integer number, which in CIF is represented by the NatType.

\[
\begin{align*}
\text{ArrayType} & \xrightarrow{X} \text{ArrayType} \\
\text{elementType} & = e'_{et} \\
\text{dimension} & = e'_{d}
\end{align*}
\]

6.1.2 BooleanType

Because it has no subexpressions, every instance of the BooleanType constant is a valid type.

\[
\text{BooleanType} \xrightarrow{(\text{}} \text{BooleanType} \\
() \\
(}
\]

6.1.3 DictionaryType

Dictionaries contain keys with associated values. In DictionaryType, this is represented by two properties, keyType and valueType. Both need to be valid types.
\[
\begin{align*}
  e_{kt} & \xrightarrow{X} e'_{kt} \\
  e_{vt} & \xrightarrow{X} e'_{vt}
\end{align*}
\]

\[\text{DictionaryType} \quad (\text{keyType} = e_{kt} \quad \text{valueType} = e_{vt}) \xrightarrow{X} \text{DictionaryType} \quad (\text{keyType} = e'_{kt} \quad \text{valueType} = e'_{vt})\]

### 6.1.4 EnumType

As long as all constants that are part of an enumeration are proper constants, which is decided by the parser, all enumerations are proper types.

\[
\forall e : \forall f : (e = f) \Rightarrow (e\text{.name} = f\text{.name})
\]

\[
\text{EnumType} \quad (\text{elements} = e^{*}) \implies \text{EnumType} \quad ()
\]

### 6.1.5 IntType

As with \text{BoolType}, every instance of \text{IntType} is a valid type.

\[
\text{IntType} \quad () \implies \text{IntType} \quad ()
\]

### 6.1.6 ListType

Lists contain a variable number of elements of the same type. This is represented \text{ListType} by the \text{elementType} property, which must contain a valid type.

\[
\begin{align*}
  e_{et} & \xrightarrow{X} e'_{et} \\
  e_{r} & \xrightarrow{X} e'_{r}
\end{align*}
\]

\[
\text{ListType} \quad (\text{elementType} = e_{et}) \xrightarrow{X} \text{ListType} \quad (\text{elementType} = e'_{et})
\]

### 6.1.7 MatrixType

Matrices are used in CIF to represent a number of numerical values. Matrices always are two-dimensional, so \text{MatrixType} has two properties, rows and columns, that must contain natural numbers to describe the size. The final property \text{elementType} must contain a valid type. Note that we do not test here if the type actually represents numerical data, because there is only one kind of type, so we cannot distinguish numerical types.

\[
\begin{align*}
  e_{et} & \xrightarrow{X} e'_{et} \\
  e_{r} & \xrightarrow{X} e'_{r}
\end{align*}
\]

\[
\text{MatrixType} \quad (\text{elementType} = e_{et} \quad \text{rows} = e_{r} \quad \text{columns} = e_{c}) \xrightarrow{X} \text{MatrixType} \quad (\text{elementType} = e'_{et} \quad \text{rows} = e'_{r} \quad \text{columns} = e'_{c})
\]

24
6.1.8 NatType
Like BoolType and IntType, all instances of NatType are valid types.

\[
\text{NatType} \rightarrow \text{NatType}
\]

6.1.9 RealType
Like BoolType, IntType and NatType, all instances of RealType are valid types.

\[
\text{RealType} \rightarrow \text{RealType}
\]

6.1.10 SetType
Like lists, sets contain a variable number of values of identical types. Thus, like with ListType, SetType has a property `elementType` that has to contain a valid type.

\[
e_{et} \xrightarrow{X} e'_{et} \\
\text{SetType} \rightarrow \text{SetType}
\]

6.1.11 TypeVariable
Type variable elements represent a type declared elsewhere, for example as part of an invocation of a function with type parameters.

\[
\text{lookUp}(n, TENV) = t_1 \\
\text{ReferenceExpression} \rightarrow \text{TypeVariable}
\]

6.1.12 VectorType
Like matrices, vectors in CIF contain numeric values, but this time they have only one dimension, represented by the `dimension` property, that must contain a natural number as value. Like all collections, there is also a property `elementType` that describes the type of the elements. Like in the MatrixType case, we check that it contains a valid type, but not that the type represents numeric values.

\[
e_{et} \xrightarrow{X} e'_{et} \\
e_d \xrightarrow{X} e'_d \\
\text{lubupdate}(e'_{d,\text{type}} : \text{NatType}, T) = \text{NatType}, T_1 \\
\text{VectorType} \rightarrow \text{VectorType}
\]

7 Expressions
7.1 Constants
Constants are used to represent basic values in CIF models.
7.1.1 BoolLiteral

Elements of type `BoolLiteral` represent basic boolean values, i.e. true and false. Because the metamodel guarantees those are the only values the elements can contain, all elements get the type `BoolType`.

\[
\text{BoolLiteral} () \implies \text{BoolLiteral} ( \text{type} = \text{BoolType} )
\]

7.1.2 Number

Elements of type `Number` represent basic non-negative integer values. Because the metamodel guarantees all values the elements can contain are integer numbers, elements of this type can get all numeric types. This is represented by a type variable, because we do not know at this point which type fits with the rest of the model.

\[
\text{Number} () \xrightarrow{\text{typename} = T, \text{typename}' = T2, X} \text{Number} ( \text{type} = \text{TypeVariable} ( \text{name} = I ) )
\]

7.1.3 RealLiteral

Elements of type `RealLiteral` represent numeric values. Because the metamodel guarantees those are the only values the elements can contain, all elements get the type `RealType`.

\[
\text{RealNumber} () \implies \text{RealNumber} ( \text{type} = \text{RealType} )
\]

7.1.4 StringLiteral

Elements of type `StringLiteral` represent string values. Because the metamodel guarantees those are the only values the elements can contain, all elements get the type `StringType`.

\[
\text{StringLiteral} () \implies \text{StringLiteral} ( \text{type} = \text{StringType} )
\]

7.1.5 TimeLiteral

Elements of type `TimeLiteral` represent numeric values that represent times. Because the metamodel guarantees those are the only values the elements can contain, all elements get the type `RealType`.

\[
\text{TimeLiteral} () \implies \text{TimeLiteral} ( \text{type} = \text{RealType} )
\]
7.2 BinaryExpression

One of the main ways to combine expressions into one expression are binary expressions. A binary expression applies a binary operator to two subexpressions, commonly called left and right. Because of the number of binary operators involved, the rules are split up into several sections.

7.2.1 General Binary Expressions

Most binary expressions can be handled by this rule, because they only involve basic types. There is a list of operators with potential types, represented by the getoptypes, in which we can look up the possible types. We can then compare the types of the subexpressions to the parameter types of the operator, which then gives us the possible result types.

\[
\begin{align*}
O &= \text{getOpTypes}(o) \\
\text{lhs} &\xrightarrow{X} \text{lhs}' \\
\text{rhs} &\xrightarrow{X} \text{rhs}' \\
\text{addConstraint}(I\in O, T1) &= T2 \\
\text{addParameterConstraints}(I\in \text{parameters}, \text{lhs'} : \text{rhs'}, T2) &= T3
\end{align*}
\]

<table>
<thead>
<tr>
<th>BinaryExpression</th>
<th>BinaryExpression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\text{operator} = o, \text{leftChild} = \text{lhs}, \text{rightChild} = \text{rhs}) )</td>
<td>( (\text{operator} = o, \text{leftChild} = \text{lhs}', \text{rightChild} = \text{rhs}', \text{type} = I.\text{returnType}) )</td>
</tr>
</tbody>
</table>

7.2.2 Concatenation

When two container values are concatenated, we have to compare not only the container type but also the type of the elements. Thus, this operation has a separate set of rules where we can add these extra preconditions.

**Array** Arrays are a form of collection with a set number of elements. Thus, if we want to combine two arrays, we need to not only ensure the elements are all of the same type, but also construct a new array type that is the right size to contain all elements. Because we know the sizes of the arrays involved in advance, we do not need a constraint to compute the length, but it can be computed directly.

\[
\begin{align*}
\text{lhs} &\xrightarrow{X} \text{lhs}' \\
\text{addConstraint}(\text{lhs'}:\text{type} = \text{ArrayType}(\text{elementType} = t1, \text{dimension} = n1), T) &= T1 \\
\text{rhs} &\xrightarrow{X} \text{rhs}' \\
\text{addConstraint}(\text{rhs'}:\text{type} = \text{ArrayType}(\text{elementType} = t2, \text{dimension} = n2), T1) &= T2 \\
\text{lubupdate} (t1 : t2, T2) &= t3, T3 \\
n_3 &= n1 + n2
\end{align*}
\]

<table>
<thead>
<tr>
<th>BinaryExpression</th>
<th>BinaryExpression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\text{operator} = \text{Concatenation}, \text{leftChild} = \text{lhs}, \text{rightChild} = \text{rhs}) )</td>
<td>( (\text{operator} = \text{Concatenation}, \text{leftChild} = \text{lhs}', \text{rightChild} = \text{rhs}', \text{type} = \text{ArrayType}(\text{elementType} = t3, \text{dimension} = n3)) )</td>
</tr>
</tbody>
</table>
**List**  A list is the one of the simpler containers to type, because it has no size to keep track of. The rule only needs to compare the element types of the subexpressions, to ensure the combined list can contain all elements.

\[
\begin{align*}
\text{lhs} & \xrightarrow{X} \text{lhs}' \\
\text{addConstraint}((\text{lhs}'.\text{type} = \text{ListType}(&\text{elementType} = t_1), T) &= T1 \\
\text{rhs} & \xrightarrow{X} \text{rhs}' \\
\text{addConstraint}((\text{rhs}'.\text{type} = \text{ListType}(&\text{elementType} = t_2), T1) &= T2 \\
\text{lubupdate} (t_1 : t_2, T) &= t_3, T1
\end{align*}
\]

**String**  Strings are a special case for the operator. Despite not being a container in the CIF sense, strings can be concatenated together. Because strings have no elements nor set lengths, the only thing we need to check is that both subexpressions have type `StringType`.

\[
\begin{align*}
\text{lhs} & \xrightarrow{X} \text{lhs}' \\
\text{addConstraint}((\text{lhs}'.\text{type} = \text{StringType}, T) &= T1 \\
\text{rhs} & \xrightarrow{X} \text{rhs}' \\
\text{addConstraint}((\text{rhs}'.\text{type} = \text{StringType}, T) &= T1
\end{align*}
\]

**Tuple**  Tuples are the only CIF container that can contain elements of differing types. Tuples have fields, that optionally can be named. In order to concatenate two tuples together, we have to combine the two sets of fields. This is handled by the `mergefields` function. If the tuples have unnamed fields, they are concatenated together like a list. If there are named fields, fields that share names are combined, where the left subexpression has precedence. Finally, a tuple type without field names is constructed.

\[
\begin{align*}
\text{lhs} & \xrightarrow{X} \text{lhs}' \\
\text{lhs}'.\text{type} & = \text{TupleType}(\text{fields} = f_1) \\
\text{rhs} & \xrightarrow{X} \text{rhs}' \\
\text{rhs}'.\text{type} & = \text{TupleType}(\text{fields} = f_2) \\
f_3 & = \text{mergeFields}(f_1, f_2)
\end{align*}
\]

**Vector**  Vectors are a special type of type of list for numeric elements. Like list, we need to ensure all elements are of the same type, but also that the resulting element type is a subtype of `RealType`. 

---

28
lhs \xrightarrow{X} lhs'

lhs'.type = VectorType(elementType = t_1, dimension = n_1)

rhs \xrightarrow{X} rhs'

rhs'.type = VectorType(elementType = t_2, dimension = n_2)

lubupdate(t_1 : t_2, T) = t_3, T_1

lubupdate(t_3 : RealType, T_1) = t_4, T_2

7.2.3 ElementTest

When we have a container and a value, we can test whether the value occurs in the container. This operator always returns a value of BoolType.

Array For this operation, we need to ensure that the type of the elements of the container matches the type of the value to be checked. Because the number of elements does not matter, we need not take the size of the array into account. The resulting rule is very similar to those of lists and vectors.

lhs \xrightarrow{X} lhs'

rhs \xrightarrow{X} rhs'

rhs'.type = ArrayType(elementType = t_2)

lubupdate(lhs'.type : t_2, T) = t_3, T_1

BinaryExpression

(operator = ElementTest
leftChild = lhs
rightChild = rhs) \xrightarrow{X}

(BinaryExpression
(operator = ElementTest
leftChild = lhs'
rightChild = rhs')

(type = VectorType
(elementType = t_4
dimension = n_1 + n_2))

Dictionary When the ElementTest operator is applied to a dictionary, it can be used to determine it is occurs as a key. This means the rule has to check whether the type of the value matches the type of the key, the type of the elements is irrelevant.

lhs \xrightarrow{X} lhs'

rhs \xrightarrow{X} rhs'

rhs'.type = DictionaryType(keyType = t_k)

lubupdate(lhs'.type : t_k, T) = t_3, T_1

BinaryExpression

(operator = ElementTest
leftChild = lhs
rightChild = rhs) \xrightarrow{X}

(BinaryExpression
(operator = ElementTest
leftChild = lhs'
rightChild = rhs')

(type = BoolType)
List  Because lists are such basic containers, the only thing the rule needs to ensure is that the
type of the value to be checked matches the type of the elements.

\[
\begin{align*}
\text{lhs} & \xrightarrow{X} \text{lhs}' \\
\text{rhs} & \xrightarrow{X} \text{rhs}' \\
\text{rhs}'.\text{type} & = \text{ListType}(\text{elementType} = t_2) \\
\text{lubupdate}(\text{lhs}.\text{type} : t_2, T) & = t_3, T_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>BinaryExpression</th>
<th>BinaryExpression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{operator} = \text{ElementTest})</td>
<td>(\text{operator} = \text{ElementTest})</td>
</tr>
<tr>
<td>(\text{leftChild} = \text{lhs})</td>
<td>(\text{leftChild} = \text{lhs}')</td>
</tr>
<tr>
<td>(\text{rightChild} = \text{rhs})</td>
<td>(\text{rightChild} = \text{rhs}')</td>
</tr>
<tr>
<td>(\text{type} = \text{BoolType})</td>
<td>(\text{type} = \text{BoolType})</td>
</tr>
</tbody>
</table>

Set  Like in the case where \text{ElementTest} is applied to a list, the rule only needs to ensure that
the type of the value matches the type of the elements.

\[
\begin{align*}
\text{lhs} & \xrightarrow{X} \text{lhs}' \\
\text{rhs} & \xrightarrow{X} \text{rhs}' \\
\text{rhs}'.\text{type} & = \text{SetType}(\text{elementType} = t_2) \\
\text{lubupdate}(\text{lhs}.\text{type} : t_2, T) & = t_3, T_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>BinaryExpression</th>
<th>BinaryExpression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{operator} = \text{ElementTest})</td>
<td>(\text{operator} = \text{ElementTest})</td>
</tr>
<tr>
<td>(\text{leftChild} = \text{lhs})</td>
<td>(\text{leftChild} = \text{lhs}')</td>
</tr>
<tr>
<td>(\text{rightChild} = \text{rhs})</td>
<td>(\text{rightChild} = \text{rhs}')</td>
</tr>
<tr>
<td>(\text{type} = \text{BoolType})</td>
<td>(\text{type} = \text{BoolType})</td>
</tr>
</tbody>
</table>

Vector  When applied to a vector, the value to be checked must be numeric, because vectors
can only contain numeric values. However, because we can be sure a \text{VectorType} element has a
numeric type as \text{elementType}, the rule only needs to check if the types match.

\[
\begin{align*}
\text{lhs} & \xrightarrow{X} \text{lhs}' \\
\text{rhs} & \xrightarrow{X} \text{rhs}' \\
\text{rhs}'.\text{type} & = \text{VectorType}(\text{elementType} = t_2) \\
\text{lubupdate}(\text{lhs}.\text{type} : t_2, T) & = t_3, T_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>BinaryExpression</th>
<th>BinaryExpression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{operator} = \text{ElementTest})</td>
<td>(\text{operator} = \text{ElementTest})</td>
</tr>
<tr>
<td>(\text{leftChild} = \text{lhs})</td>
<td>(\text{leftChild} = \text{lhs}')</td>
</tr>
<tr>
<td>(\text{rightChild} = \text{rhs})</td>
<td>(\text{rightChild} = \text{rhs}')</td>
</tr>
<tr>
<td>(\text{type} = \text{BoolType})</td>
<td>(\text{type} = \text{BoolType})</td>
</tr>
</tbody>
</table>

7.2.4 Equal & NotEqual

As one would expect, the operators to test for the equality or inequality of two values are typed
in a similar way. According to the CIF semantics, values can only be compared if their types are
equal, so that is the condition the rule uses.
7.2.5 Intersection & SetSubtraction & Union

Unlike lists and arrays, sets cannot be concatenated. Instead, they can be combined in several other ways. In all cases, we have to compare the types of the elements, and create a new set that can contain all elements.

\[ \alpha\{\text{Intersection, SetSubtraction, Union}\} \]
\[ lhs \xrightarrow{=} lhs' \]
\[ rhs \xrightarrow{=} rhs' \]
\[ lubupdate(lhs'.\text{type} : rhs'.\text{type}, T) = t, T' \]

7.2.6 ListSubtraction

In addition to concatenation, we can combine two lists by subtracting one from the other. From the typing perspective, the result is the same: the rule has to ensure that the types of the elements match.

\[ \alpha\{\text{ListSubtraction}\} \]
\[ lhs \xrightarrow{=} lhs' \]
\[ rhs \xrightarrow{=} rhs' \]
\[ lhs'.\text{type} = \text{ListType}(\text{elementType} = t_1) \]
\[ rhs'.\text{type} = \text{ListType}(\text{elementType} = t_2) \]
\[ lubupdate(t_1 : t_2, T) = t_3, T_1 \]

7.2.7 Projection

The projection operator can be used to retrieve values from containers. The type of the value retrieved obviously matches that of the type of the elements of the container, what changes per container type is what we need to retrieve a value.
**Array** Values can be retrieved from an array by giving their index. This index has to be of type `NatType`, i.e., a positive, integer number, because arrays never contain elements with negative or noninteger indexes. Note that because it is not required that the actual index value is known at compile-time, we do not statically check whether the index value provided falls within the bounds of the array.

\[
\begin{align*}
\text{BinaryExpression} & \quad \text{BinaryExpression} \\
\left( \begin{array}{c}
\text{operator} = \text{Projection} \\
\text{leftChild} = \text{lhs} \\
\text{rightChild} = \text{rhs}
\end{array} \right) & \quad \xrightarrow{X} \quad 
\left( \begin{array}{c}
\text{operator} = \text{Projection} \\
\text{leftChild} = \text{lhs'} \\
\text{rightChild} = \text{rhs'} \\
\text{type} = t_e
\end{array} \right)
\end{align*}
\]

\[
\text{lubupdate} \left( \text{rhs'}.\text{type} : \text{NatType}, T \right) = \text{NatType}, T_1
\]

**Dictionary** When retrieving an element from a dictionary, we have to provide a key value that is in the dictionary and get its corresponding value. Thus, the rule ensures that the type of the value provided matches hat of the type of the keys, and the type of the result matches the type of the elements.

\[
\begin{align*}
\text{BinaryExpression} & \quad \text{BinaryExpression} \\
\left( \begin{array}{c}
\text{operator} = \text{Projection} \\
\text{leftChild} = \text{lhs} \\
\text{rightChild} = \text{rhs}
\end{array} \right) & \quad \xrightarrow{X} \quad 
\left( \begin{array}{c}
\text{operator} = \text{Projection} \\
\text{leftChild} = \text{lhs'} \\
\text{rightChild} = \text{rhs'} \\
\text{type} = t_v
\end{array} \right)
\end{align*}
\]

\[
\text{lubupdate} \left( \text{rhs'}.\text{type} : t_k, T \right) = t_k, T_1
\]

**Tuple by index** Like with arrays, the fields of a tuple can be accessed by providing their index. For tuples with unnamed fields, this is the only way of retrieving their values. Because the type of the returned value depends on the index, we need to compute it here using the eval function to be able to type this operation. As with arrays, the value that is used as index must be a positive, integer number.

\[
\begin{align*}
\text{BinaryExpression} & \quad \text{BinaryExpression} \\
\left( \begin{array}{c}
\text{operator} = \text{Projection} \\
\text{leftChild} = \text{lhs} \\
\text{rightChild} = \text{rhs}
\end{array} \right) & \quad \xrightarrow{X} \quad 
\left( \begin{array}{c}
\text{operator} = \text{Projection} \\
\text{leftChild} = \text{lhs'} \\
\text{rightChild} = \text{rhs'} \\
\text{type} = f_v.\text{type}
\end{array} \right)
\end{align*}
\]

\[
\text{lubupdate} \left( \text{rhs'}.\text{type} : \text{NatType}, T \right) = \text{NatType}, T_1
\]

\[
\text{eval}(\text{rhs'}) = v \\
0 \leq v < n
\]
**Tuple by name**  Tuples are a special case, in that there are two ways to retrieve values from them. The second option is to refer to fields by their name. This can obviously only be done if the field has a name. The `rightChild` must contain a `ReferenceExpression`, that is typed in an environment that consist only of fields. If typing succeeds, the resulting `FieldReference`, has the type of the field referenced, which is also the result of the projection.

\[
\text{lhs}' = \text{FieldReference}(\text{type} = \text{t})
\]

**Vector**  Elements in a vector can be accessed using by providing their index. The index must be of type `NatType`, i.e. a positive, integer number, because vectors never contain elements with negative or noninteger indexes. Note that, like with arrays, we do not check if the index falls within the bounds of the vector.

\[
\text{lhs}' = \text{FieldReference}(\text{type} = \text{t})
\]

**7.2.8 Subset**

For sets, we can determine whether all elements of one set also occur in the second. In order to compare elements, they need to be of equal types, so that is the main precondition of this rule. If the types match, the result is always of type `BoolType`.

\[
\text{lhs}' = \text{FieldReference}(\text{type} = \text{t})
\]

33
7.3 UnaryExpression

In addition to binary expression, with two subexpression, there are also unary expressions, with only one subexpression, called leftChild. Though there are less unary operators than binary operators, they are still split into separate sections.

7.3.1 Derivative

The derivative operator computes, as the name suggest, the derivative of an expression. It is unusual, in that this operator only applies to specific kinds of expressions.

Variable Variable references are one kind of expression from which derivatives can be calculated. The result is always a value of RealType, independent of the type of the actual variable. For non-numeric variables, the result will always be zero.

\[ lhs \xrightarrow{X} lhs' \]

\[ lhs' = \text{VariableReference}(\) \]

Clock Clock references are another type of expression from which derivatives can be calculated. The resulting type is the same as that of the clock referenced.

\[ lhs \xrightarrow{\text{CENV}} lhs' \]

\[ lhs' = \text{ClockReference}(\) \]

7.3.2 Inverse & Negate

there are two unary functions defined as part of getoptypes, the inverse and the negate operator. The rule compares the type of the argument to the type of the parameter, and if it matches the result is the returnType of the operator.

\[ O = \text{getOpTypes}(a) \]

\[ lhs \xrightarrow{X} lhs' \]

\[ \text{addConstraint}((\leq, O, T1) = T2) \]

\[ \text{addParameterConstraints}(I, \text{type}, \text{parameters}, lhs', T2) = T3 \]

7.3.3 New

The new operator can be used to compute the next value of a continuous value. Like the derivative operator, it can only be applied to a limited set of expressions.
**Variable**  When applied to a variable, the \texttt{new} operator returns a value of the same type as the variable.

\[
\begin{align*}
\text{lhs} & \xrightarrow{VENV} \text{lhs}' \\
\text{lhs}' & = \text{VariableReference}()
\end{align*}
\]

When applied to a variable, the \texttt{new} operator returns a value of the same type as the variable.

\[
\begin{align*}
\text{lhs} & \xrightarrow{VENV} \text{lhs}' \\
\text{lhs}' & = \text{VariableReference}()
\end{align*}
\]

\[
\begin{align*}
\text{UnaryExpression} & \quad \text{UnaryExpression} \\
\left( \text{operator} = \text{New} \right) & \quad \left( \text{operator} = \text{New} \right) \\
\left( \text{leftChild} = \text{lhs} \right) & \quad \left( \text{leftChild} = \text{lhs}' \right) \\
\text{varenv} & \xrightarrow{VENV.X} \text{lhs}' \\
\text{lhs}' & = \text{lhs}'.\text{type}
\end{align*}
\]

**Clock**  When applied to a clock, the \texttt{new} operator returns a value of the same type as the clock.

\[
\begin{align*}
\text{lhs} & \xrightarrow{CENV} \text{lhs}' \\
\text{lhs}' & = \text{ClockReference}()
\end{align*}
\]

When applied to a clock, the \texttt{new} operator returns a value of the same type as the clock.

\[
\begin{align*}
\text{UnaryExpression} & \quad \text{UnaryExpression} \\
\left( \text{operator} = \text{New} \right) & \quad \left( \text{operator} = \text{New} \right) \\
\left( \text{leftChild} = \text{lhs} \right) & \quad \left( \text{leftChild} = \text{lhs}' \right) \\
\text{clockenv} & \xrightarrow{CENV.X} \text{lhs}' \\
\text{lhs}' & = \text{lhs}'.\text{type}
\end{align*}
\]

**Derivative**  When applied to the derivative of variable, the \texttt{new} operator returns a value of the same type as the derivative.

\[
\begin{align*}
\text{lhs} = \text{UnaryExpression}(\text{operator} = \text{Derivative}, \text{leftChild} = l) \\
\text{lhs} & \xrightarrow{l} \text{lhs}'
\end{align*}
\]

When applied to the derivative of variable, the \texttt{new} operator returns a value of the same type as the derivative.

\[
\begin{align*}
\text{UnaryExpression} & \quad \text{UnaryExpression} \\
\left( \text{operator} = \text{New} \right) & \quad \left( \text{operator} = \text{New} \right) \\
\left( \text{leftChild} = \text{lhs} \right) & \quad \left( \text{leftChild} = \text{lhs}' \right) \\
\text{lhs} & \xrightarrow{X} \text{lhs}' \\
\text{lhs}' & = \text{lhs}'.\text{type}
\end{align*}
\]

**7.3.4 Pick**

The \texttt{Pick} operator can be used to get an arbitrary element from a non-empty container. As such, it results in a value of the same type as the elements of the container that it is applied to.

\[
\begin{align*}
\text{lhs} & \xrightarrow{X} \text{lhs}' \\
\text{lhs}'.\text{type} & = \text{ContainerType}(\text{elementType} = t)
\end{align*}
\]

**7.4 ArrayExpression**

Elements of type \texttt{ArrayExpression} are the basic way to construct arrays in CIF. An \texttt{ArrayExpression} element with \(n\) subexpressions creates an array of size \(n\), and all subexpressions must have the same type, \(t_e\).
\[
\begin{align*}
\text{element}^* \xrightarrow{X} \text{element1}^* \\
\text{lubupdate}(\text{element1}^*.\text{type}, T) = t_e, T1
\end{align*}
\]

\[
\begin{array}{l}
\text{ArrayExpression} \\
( \text{elements} = \text{element}^* ) \\
\xrightarrow{X} \\
\text{ArrayExpression} \\
( \text{elements} = \text{element1}^* ) \\
\text{ArrayType} \\
( \text{elementType} = t_e ) \\
( \text{dimension} = n )
\end{array}
\]

### 7.5 ClockReference

In CIF, clocks are special entities that indicate the passing of time. As such, there is a special type of reference dedicated to them, `ClockReference`. Clocks are always of type `RealType`, but that is checked as part of the declaration of the clock.

\[
\text{lookUp}(n, \text{CENV}) = c_1
\]

\[
\begin{array}{l}
\text{ReferenceExpression} \\
( \text{name} = n ) \\
\xrightarrow{CENV.X} \\
\text{ClockReference} \\
( \text{type} = c_1.\text{type} ) \\
( \text{clock} = c_1 )
\end{array}
\]

### 7.6 ConditionalAlternative

Elements of type `ConditionalAlternative` only occur as subexpressions of `ConditionalExpression` elements. Each `ConditionalAlternative` represents one case, and consists of a subexpression that serves as guard and a subexpressions that provides the value. The first two preconditions check whether the guard can be typed as has the correct type, while the last precondition checks whether the value expressions can be typed.

\[
g \xrightarrow{X} g' \\
\text{lubupdate} (g'.\text{type} : \text{BoolType}, T) = \text{BoolType}, T1 \\
e \xrightarrow{X} e'
\]

\[
\begin{array}{l}
\text{ConditionalAlternative} \\
( \text{guard} = g ) \\
( \text{expression} = e ) \\
\xrightarrow{X} \\
\text{ConditionalAlternative} \\
( \text{guard} = g' ) \\
( \text{expression} = e' )
\end{array}
\]

### 7.7 ConditionalExpression

Each `ConditionalExpression` elements consists of a number of `typenameConditionalAlternative` elements. Each of these subexpressions must be typed correctly, according to the first precondition. The second precondition checks that all alternatives have the same type, so the expression is guaranteed to return an element of that type no matter which option is chosen.
7.8 DictionaryExpression

Elements of type DictionaryExpression are the basic way to create dictionaries in CIF. Because empty dictionaries are allowed, there are two rules to handle this special case.

7.8.1 Empty

This rule defines the type of empty DictionaryExpression elements. Both the type for the keys and the values of the dictionary are set to constraint variables, because empty dictionaries are valid for any kind of dictionary.

\[
\text{DictionaryExpression} \quad \text{pairs} = \varepsilon \quad \Rightarrow \quad \text{DictionaryExpression} \quad \text{pairs} = \varepsilon \\
\text{DictionaryType} \quad \text{type} = \begin{cases} 
\text{TypeVariable} & \text{name} = I \\
\text{TypeVariable} & \text{name} = J 
\end{cases}
\]

7.8.2 Nonempty

A DictionaryExpression element that is not empty has a number of DictionaryPair subexpressions. The non-emptyness of the rule is checked in the first line of the condition. Next, the rule checks if all subexpressions can be typed. If that is the case, the types of keys and values are extracted and compared to ensure they are the same for each pair. The resulting types are used to construct the result DictionaryType.

\[
\text{length(pair*)} > 0 \\
\text{pair*} \quad \Rightarrow \quad \text{pair1*} \\
\text{lubupdate}(k*.type, T) = t_k, T1 \\
\text{lubupdate}(v*.type, T1) = t_v, T2
\]

\[
\text{DictionaryExpression} \quad \text{pairs} = \text{pair*} \quad \Rightarrow \quad \text{DictionaryExpression} \quad \text{pairs} = \text{pair*} \\
\text{DictionaryType} \quad \text{type} = \begin{cases} 
\text{TypeVariable} & \text{name} = t_k \\
\text{TypeVariable} & \text{name} = t_v 
\end{cases}
\]

7.9 DictionaryPair

DictionaryPair elements occur as part of DictionaryExpression elements. Each pair consists of a key and a value, and both have to be typed successfully in order to correctly type the
DictionaryPair element.

\[
\begin{align*}
  k & \overset{X}{\rightarrow} k_1 \\
v & \overset{X}{\rightarrow} v_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>DictionaryPair</th>
<th>DictionaryPair</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\text{key} = k) )</td>
<td>( (\text{key} = k_1) )</td>
</tr>
<tr>
<td>( (\text{value} = v) )</td>
<td>( (\text{value} = v_1) )</td>
</tr>
</tbody>
</table>

7.10 Distribution

Elements of type Distribution refer to one of the predefined probability distributions. There may be multiple defined distributions with the same name. First, the rule checks what possible distributions for the given name exist. The next precondition checks whether the arguments can be typed correctly. The third precondition states that the chosen distribution must be one of those selected, and the final precondition states that the types of the arguments must match the type of the parameters of the chosen distribution.

\[
\begin{align*}
D &= \text{getDistributionTypes}(n) \\
a^* &\overset{X}{\rightarrow} a^* \\
\text{addConstraint}(I \in D, T) = T_1 \\
\text{addParameterConstraints}(I\text{.type.parameters}, a^*, T_1) = T_2 \\
s &\overset{X}{\rightarrow} s_1 \\
\text{lubupdate}(s\text{.type : NatType, } T_2) = \text{NatType, } T_3
\end{align*}
\]

7.11 FieldReference

FieldReference elements only occur as part of projections by name on tuples. In such circumstance, the environment contains a list of fields, and this rule selects the appropriate one based on its name.

\[
\text{lookUp}(n, FENV) = f_1
\]

\[
\begin{align*}
\text{ReferenceExpression} &\overset{FENV}{\rightarrow} \text{FieldReference} \\
(\text{name} = n_1) &\overset{FENV}{\rightarrow} (\text{function} = f_1, \text{type} = f_1\text{.type})
\end{align*}
\]

7.12 FunctionCallExpression

Elements of type FunctionCallExpression consist of one subexpression that describes the function to be typed, and a set of subexpressions that provide values for the parameters. The first precondition checks if the function subexpression can be correctly typed. The next checks the same for each of the arguments. The third creates a constraint for each parameter-argument pair that enforces that their types must match, so a function can only be applied if the types of all arguments match the types of their parameter. The resulting type is the type returned by the function.
\[ f \xrightarrow{X} f' \]
\[ a^* \xrightarrow{X} a1^* \]
\[ \text{addParameterConstraints}(f', \text{type}, \text{formalParameters}, a1^*, T) = T1 \]

\[ \begin{array}{l}
\text{FunctionCallExpression} \\
\quad ( \text{function} = f ) \\
\quad ( \text{arguments} = a^* ) \\
\quad \text{StdLibenv} = \text{LENV} \times \\
\quad \rightarrow \\
\end{array} \]

\[ \begin{array}{l}
\text{FunctionCallExpression} \\
\quad ( \text{function} = f' ) \\
\quad ( \text{arguments} = a1^* ) \\
\quad ( \text{type} = f'.\text{type}.\text{returntype} ) \\
\end{array} \]

### 7.13 FunctionReference

Elements of type `FunctionReference` describe references to functions. The rule looks up which functions can be referenced in the environment, and a constraint is created to state one of these must be chosen.

\[ \text{lookUp}(F, FENV) = f_1 \]
\[ \text{addConstraint}(IeF, T) = T1 \]

\[ \begin{array}{l}
\text{ReferenceExpression} \\
\quad ( \text{name} = n_1 ) \\
\quad \text{functionenv} = \text{FENV} \times \\
\quad \rightarrow \\
\end{array} \quad \begin{array}{l}
\text{FunctionReference} \\
\quad ( \text{function} = f_1 ) \\
\quad ( \text{type} = \text{TypeVariable} ) \\
\end{array} \]

### 7.14 GlobalConstantReference

Elements of type `GlobalConstantReference` describe references to global constants, constants that are declared as part of specifications. The rule looks up which constant can be referenced in the environment, and sets its type as the type of the expression. Because constants are considered equivalent to literal, they can have multiple types.

\[ \text{lookUp}(C, GCENV) = c_1 \]
\[ \text{addConstraint}(IeC, T) = T1 \]

\[ \begin{array}{l}
\text{ReferenceExpression} \\
\quad ( \text{name} = n_1 ) \\
\quad \text{gconstenv} = \text{GCENV} \times \\
\quad \rightarrow \\
\end{array} \quad \begin{array}{l}
\text{GlobalConstantReference} \\
\quad ( \text{constant} = c_1 ) \\
\quad ( \text{type} = \text{TypeVariable} ) \\
\end{array} \]

### 7.15 LambdaExpression

`LambdaExpression` elements allow functions to be defined as part of expressions. In order to type `LambdaExpression` elements, we first have to type its parameter definitions. These parameters definitions can then be used to fill the parameter component of the environment. Then the expression is typed, with only the parameters, global constants and standard library in the environment, because references in lambda expressions are restricted to elements of those components. Finally, a function type is constructed based on the types of the parameters and the type of the expression.

39
parameter \text{GCENV}, \text{SFENV}, X \Rightarrow \text{parameter1}^* \\
update(\{\}, \text{parameter1}^*) = \text{PENV} \\
e \text{gconstenv} = \text{GCENV}, \text{stdlibfunctionenv} = \text{SFENV}, \text{parameterenv} = \text{PENV} \ e'$

\[
\lambda e \text{parameter1}^* \\
\text{returnExpression} = e \\
g\text{constenv} = \text{GCENV}, \text{stdlibfunctionenv} = \text{SFENV}, X
\]

\[
\lambda \text{formalParameters} = \text{parameter1}^* \\
\text{returnExpression} = e' \\
\text{type} = \text{FunctionType} \\
\left( \text{formalParameters} = \text{parameter1}^* \text{. type} \right) \\
\text{returnType} = e'. \text{type}
\]

7.16 ListExpression

Elements of type ListExpression are the basic way to create lists in CIF. Lists can be empty, which creates a special case that is handled in a separate rule.

7.16.1 Empty

If a ListExpression element has no subexpressions, we cannot base the value of the \text{elementType} of the \text{ListType} the rule creates on the types of the subexpressions. Instead, we insert only a type variable, relying on the rest of the model to provide further constraints.

\[
\text{ListExpression} \left( \text{elements} = \varepsilon \right) \xrightarrow{X} \left( \begin{array}{c}
\text{elements} = \varepsilon \\
\text{type} = \text{ListType} \left( \text{elementType} = \text{TypeVariable} \left( \text{name} = I \right) \right)
\end{array} \right)
\]

7.16.2 Nonempty

If the ListExpression has subexpressions, we first check if those subexpressions can be typed. If that is the case, the next condition checks if the types of all subexpressions can be made equal to one type, \(t_c\). This value is then used to create a ListType element.

\[
\text{length} ( \text{element}^* ) > 0 \\
\text{element}^* \xrightarrow{X} \text{element1}^* \\
\text{lubupdate} ( \text{element1}^* . \text{type}, T ) = t_c, T_1
\]

\[
\text{ListExpression} \left( \text{elements} = \text{element}^* \right) \xrightarrow{X} \left( \begin{array}{c}
\text{elements} = \text{element1}^* \\
\text{type} = \text{ListType} \left( \text{elementType} = t_c \right)
\end{array} \right)
\]

7.17 LocalConstantReference

Elements of type LocalConstantReference describe references to local constants, constants that are declared as part of specification components like models and functions. The rule looks up which constant can be referenced in the environment, and sets its type as the type of the expression. Because constant references are considered equivalent to literals, they can have multiple types.
\[ \text{lookUp}(C, LCENV) = c_1 \]
\[ \text{addConstraint}(I \epsilon C, T) = T_1 \]

\[
\begin{align*}
\text{ReferenceExpression} & \quad \text{LocalConstantReference} \\
( \text{name} = n_1 ) & \quad ( \text{constant} = c_1 ) \\
& \quad ( \text{type} = \text{TypeVariable} ) \quad ( \text{name} = I )
\end{align*}
\]

7.18 MatrixExpression

MatrixExpression elements are the basic way to define matrices in CIF. Matrices in CIF have two dimensions of set size. In order to represent this, MatrixExpression elements contain a number of MatrixRow elements that in turn contain a number of other expressions. In order to type a MatrixExpression element elements, we first check if the elements in the rows can be typed. Then, the rule checks if all rows have the same length. The next step is to check if all elements have the same type. This is done by first checking per row, and then checking the types found for each row. Finally, because matrices can contain only numeric elements, we check whether the computed typed can be widened to real.

\[
\begin{align*}
\text{row}^* & \xrightarrow{X} \text{row1}^* \\
( \forall \text{row} \in \text{row1}^* : \text{length(\text{row.columnelements})} = m) \\
T & = T_0 \\
(\forall i : \text{length(\text{row1}^*)} - 1 : \text{lubupdate(\text{row1}^*.\text{columnelements}.\text{type}, T_i) = t_{ki}, T_{i+1}}) \\
\text{lubupdate(} \text{t}_e : \text{RealType}, T_{\text{length(\text{row1}^*)}+1}) & = \text{t}_e, T_{\text{length(\text{row1}^*)}+2}
\end{align*}
\]

\[
\begin{align*}
\text{MatrixExpression} & \quad \xrightarrow{X} \\
( \text{rows} = \text{row}^* ) \\
\text{rows} & = \text{row1}^* \\
\text{type} & = \begin{pmatrix}
\text{elementType} & = \text{RealType} \\
\text{rowDimension} & = \text{length(\text{row1}^*)} \\
\text{columnDimension} & = m
\end{pmatrix}
\end{align*}
\]

7.19 MatrixRow

MatrixRow elements occur only as part of MatrixExpression elements, and do not have types themselves. As such, the only condition is that all subexpressions can be typed.

\[
\begin{align*}
\text{element}^* & \xrightarrow{X} \text{element1}^* \\
( \text{columnelements} = \text{element}^* ) & \xrightarrow{X} ( \text{columnelements} = \text{element1}^* )
\end{align*}
\]

7.20 ParameterReference

ParameterReference elements are references that target parameter declarations. They only occur inside functions and can only target parameters that have been declared as part of that particular function.
lookUp\( (n, PENV) = p_1 \)

<table>
<thead>
<tr>
<th>ReferenceExpression ( \xrightarrow{\text{parameter env}=PENV.X} ) ParameterReference</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{name} = n_1 ) ( \xrightarrow{\text{parameter} = p_1} ) ParameterReference ( \text{type} = p_1.\text{type} )</td>
</tr>
</tbody>
</table>

### 7.21 SetExpression

Elements of type `SetExpression` are the basic way to create sets in CIF. Sets can be empty, which creates a special case that is handled in a separate rule.

#### 7.21.1 Empty

If a `SetExpression` element has no subexpressions, we cannot base the value of the `elementType` of the `SetType` the rule creates on the types of the subexpressions. Instead, we insert only a type variable, relying on the rest of the model to provide further constraints.

\[
\begin{align*}
\text{SetExpression} \quad \text{elements} & = \varepsilon \\
\xrightarrow{X} & \\
\text{type} & = \text{SetType} (\text{elementType} = \text{TypeVariable} (\text{name} = I))
\end{align*}
\]

#### 7.21.2 Nonempty

If the `SetExpression` has subexpressions, we first check if those subexpressions can be typed. If that is the case, the next condition checks if the types of all subexpressions can be made equal to one type, \( t_e \). This value is then used to create a `SetType` element.

\[
\begin{align*}
\text{length} (\text{elements}^*) & > 0 \\
\text{element}^* \xrightarrow{X} & \text{element}^1 \\
\text{lubupdate} (\text{element}^1.\text{type}, T) & = t_e, T_1
\end{align*}
\]

\[
\begin{align*}
\text{SetExpression} \quad \text{elements} & = \text{element}^* \\
\xrightarrow{X} & \\
\text{type} & = \text{SetType} (\text{elementType} = t_e)
\end{align*}
\]

### 7.22 StdLibFunctionReference

`StdLibFunctionReference` elements are references to one of the predefined standard library functions. Because the functions may be overloaded, the first precondition computes the set of possible function types that are defined for the given name, and the second restricts the type to one of that set.

\[
S = \text{getStdLibTypes}(n) \\
\text{addConstraint} (I \in S, T_1) = T_2
\]

\[
\begin{align*}
\text{ReferenceExpression} \quad \text{type varenv} = T.\text{type varenv}' = T_2, X \\
\xrightarrow{\text{function} = f_1} & \\
\text{type} & = \text{TypeVariable} (\text{name} = I)
\end{align*}
\]

### 7.23 TupleExpression

`TupleExpression` elements represent basic tuple values. Unlike other containers, tuples can contain values of many different types. Values in tuples are stored in so-called fields, of which a
tuple must contain at least two. This is checked in the first precondition. The second precondition
check if the elements can be typed. The third precondition creates a number of elements to
represent the fields in the type of the TupleExpression. Note that these fields have no names.
In CIF it is not possible to create tuples with named fields using tuple expressions.

\[
\text{length}(\text{element}^*) \geq 2
\]

\[
\text{element}^* \xrightarrow{X} \text{element1}^*
\]

\[
\text{makeField}((\text{element1}^*.\text{type}) = f^*)
\]

\[
\text{TupleExpression} \xrightarrow{X} \text{TupleExpression}
\]

\[
(\text{fields} = \text{element}^*)
\]

\[
(\text{fields} = \text{element1}^*)
\]

\[
(\text{type} = \text{TupleType}(\text{fields} = f^*))
\]

### 7.24 VariableReference

VariableReference elements are references that target variable declarations. Unlike constants
the types of variables cannot be widened. Therefore, the type of the VariableReference is
exactly the type declared or the variable.

\[
\text{lookUp}(n, \text{VENV}) = v_1
\]

\[
\text{ReferenceExpression} \xrightarrow{v\text{aren}=\text{VENV}.X} \text{VariableReference}
\]

\[
(\text{name} = n_1)
\]

\[
(\text{variable} = v_1)
\]

\[
(\text{type} = v_1.\text{type})
\]

### 7.25 VectorExpression

VectorExpression elements the basic way to define vector values in CIF. Vectors consist of one
or more elements that all must have the same type. The first precondition of the rule checks
there are indeed elements in the VectorExpression. The second checks if these elements can
be typed. The third condition tests if all types can be made equal. Because vectors can only
contain numeric types, the last precondition checks if the computed type of the elements can be
widened to RealType.

\[
\text{length}(\text{element}^*) > 0
\]

\[
\text{element}^* \xrightarrow{X} \text{element1}^*
\]

\[
\text{lubupdate}(\text{element1}^*, T) = t_f, T_1
\]

\[
\text{lubupdate}(t_f : \text{RealType}, T) = \text{RealType}, T_1
\]

\[
\text{VectorExpression} \xrightarrow{X} \text{VectorExpression}
\]

\[
(\text{elements} = \text{element}^*)
\]

\[
(\text{elements} = \text{element1}^*)
\]

\[
(\text{type} = (\text{elementType} = \text{RealType})
\]

\[
(\text{dimension} = \text{length}(\text{element}^*))
\]

### 8 Related Work

The formalization of semantics, and its subset the formalization of type systems, are concepts
that have been studied for some time. Most attempts at formalization, however, focus on specific
use cases and thus their results are not easy to relate to this work. Examples of these can be
found in [1, 9, 11, 25]. In [9], a type system is described for a simple form of so-called object
models. Object models are a form of metamodels, as described in Section 2, extended with
constraints. The goal of the type system is to check the object model for constraints that are
either ambiguous or meaningless. This is done by defining a type system that permits only
expressions that are neither incorrect, ambiguous nor always empty. This work is similar to ours in that it focusses on overloading and disambiguation, but the models to be typed are much simpler and we do not consider expressions that are always true or false to be an error.

In [25], a type system is used as a method to establish properties of resource sharing protocols. In this approach, a type is defined for each kind of resource. This type is then used to check if the resource is used properly in a given protocol. In [1], types are defined for an abstract form of formulas used in spreadsheets. This work is similar to ours, in that it focusses purely on expressions, because spreadsheets usually do not have any control flow. In contrast with our approach, type widening is not limited, and functions and operations are considered purely on the abstract level, resulting of a language consisting of three constructs, namely literal numbers, references to cells and function applications, instead of dozens of constructs. In [11], a type system is described for a variation of a language used by a virtual machine. They choose not to handle concept like arithmetic expressions in order to focus on pointers, references and memory handling. This in contrast to our type system, where scoping is abstracted to focus on details of constructs. In all four of [1, 9, 11, 25], the formalism used to describe the type system are inference rules. Inference rules look and function similarly to the SOS rules described in Section 3. As can be expected based on this similarity, inference rules suffer from the same limitations. In particular, the environment has to be mentioned in full in every rule that uses it, which hinders potential language evolution. Additionally, [1, 9, 11, 25] focus on small-scale examples to provide a proof-of-concept, in contrast to the subject of this report, CIF expressions, which are part of a fully-fledged language.

Using constraints to compute types is a well-known approach. In [30], a general framework is described for typing Hindley-Milner systems [17] with constraints. This work has since been extended to cover overloading [29], annotations to support polymorph and guarded data types [28]. In [17, 29, 30], the type system is defined using inference rules that reference explicit constraint stores that form part of the environment. A similar approach is used in [15] to infer types for functions in Erlang [3], and in [2] for expressions in Ruby [10]. An unusual feature of the latter is that the type system is integrated with the dynamic semantics of Ruby, so the type system can and does use runtime information, in this case from so-called training runs, to infer types. In this work, we also use constraints in defining our type system, but as part of MSOS rules, not as part of inference rules.

A particularly interesting variation of this approach is the one proposed by Pierce et al. in [22] to define a type system for a version of the programming language ML [18]. The authors consider a statically typed language to, from the view of type inference, consist of three parts: an internal language with full type annotations, an external language with partial type annotations, and a type inference specification that relates the two languages. In this report, UCIF corresponds to the external language, CIF to the the internal languages and the type system in this document to the type inference relation. Because the external language is assumed to have some type annotations, the authors restrict themselves to local type inference, type inference using only information from adjacent nodes. This is also the approach mainly used in this document, made possible by the separation of the scoping into a separate step, so all indirect references, that would require information from non-adjacent nodes to solve, are made explicit in direct references that use information only from adjacent nodes. As in the previously mentioned type systems, the type rules are given using inference rules containing constraints. Unlike our type system, the type system in [22] is based on inference rules, not on MSOS rules, and the focus lies on function application, not on dealing with a full language.

Another related approach is described in [13], where constraints are used to compute static semantic properties like invariants and pre- and postconditions instead of types. In theory, pre- and postconditions are much stronger statements about the correctness that just type inference,
so the analysis is more accurate and the results can be used for more detailed error messages, but in practice, the constraints are much more complex and hard to solve. In fact, because the structure of the invariants and conditions is not known beforehand, second-order constraints, constraints that describe constraints, are needed to represent all possible solutions. In [13], this is solved by limiting the possible constraints to a number of predefined templates, thus limiting their power but also reducing the problem to first order constraints. In our work restrict ourselves to first-order constraints from the start, because we consider types to provide us with the required information about expressions with the extra power that can be provided by pre- and postconditions.

Two more practical approaches to type system definition are XTypeS [8], a type system DSL designed to work with the XTEXT grammar language, and XText/TS [33]. XTypeS is based on rules that consist of a conclusion and a number of preconditions. Unlike our MSOS rules, the conclusion does not involve an input and an output element, but a so-called judgement. A judgement consists of an environment and a typing statement, which in turn consists of two elements and a relation. Using these rules, the authors define relations between model elements, like subtype relations between type elements and relations between elements and types. In XTypeS typechecking is seen as validation of a model that cannot be changed, and the main goal is to provide information that editors can use to provide feedback to users. This is the main difference to our approach, where models are transformed from untyped versions to typed versions. One advantage of this is that tools using the typed version of the model can assume it is typed correctly, making them easier to build and more efficient. Any tool that accepts untyped models has to take the possibility of type errors into account, or it fails for some inputs. Another advantage is that the type information in typed models can be used by tools to improve their precision without computing additional information themselves. An interesting similarity between XTypeS and our work is that scoping is considered as a separate step, that is defined separately. In XTypeS, the scoping is defined by replacing the standard environment implementation by a custom one. This implementation can use the type system to get types of elements, and in return provides elements that belong to given names. While the requirement to implement the scoping in Java makes it less formal and convenient then we think is possible, it is an implementation of how we envision scoping in relation to the type system defined in this document. Another similarity is that type variables and simple constraints are used to represent delayed type decisions. Unlike our approach, however, type variables are not a native part of the system, but have to be added by the type system designer separately. This is because XTypeS does not use an external constraint solver like we do, but allows unification and related concepts to be defined as part of the type system explicitly.

XText/TS [33] is a type system framework designed for XTEXT expressions. XText/TS is essentially a Java library of support functions that can be used to build type systems for models based on XTEXT grammars. Like XTypeS, XText/TS type systems only validate models, they do not update the models with type information. A type system made in XText/TS is a Java class that implements the ITypeChecker interface. An editor that uses the typechecker first creates an instance of the class, and then invokes the checkTypeSystemConstraints on the model to check for errors. During initialization of the type system object, the designer can define certain elements to be types, certain features to represent types of elements and checks that verify if elements have valid types, using calls to the framework. Essentially, the framework offers a number of rule templates that can be instantiated by invoking the corresponding function. This means that only simple type system can be implemented, because the limited number of templates offered by the framework. Moreover, because the framework is implemented as a Java library, the level of abstraction is lower then what is possible in MSOS or XTypeS. An instance where this causes problems are empty lists. Due to the strict recursive nature of the
type systems implemented using this framework, a type has to be assigned to them before all relevant information is available.

9 Conclusions

We have defined a type system for the expression sublanguage of CIF. Our definition largely follows the pattern used in [23], but differs in that we relate elements to typed elements, instead of to types. This comes at a cost in succinctness, but allows us to change the structure of a model element if the computed type warrants it, like for example from general reference to a reference to a specific variable, as specified in Section 7.24, or constant, as specified in Sections 7.14 and 7.17.

The type system given in this document currently only applies to expressions. In the future, declarations and eventually entire models will have to be type checked as well. This leads, however, to ordering problems. In particular, CIF declarations can contain expressions referring to other declarations, the result of which can depend on the types and values of other expressions and declarations.

For example, array types get their bounds from expressions that in theory can be of arbitrary complexity, including references to other arrays and (overloaded) functions. If we have defined an overloaded function \( f \) that takes arrays of various sizes as arguments and returns a natural number, we could define a constant \( x \) to be of type \( \text{array}(f(\text{array}(1,1,1)),\text{int}) \). In other words, we use \( f \) to define the size of an array, introducing a dependency on \( f \) for the declaration of \( x \), but we do not know what implementation of overloaded function \( f \) we need before we start processing the declaration. In principle, these dependencies do not cause any problems if declarations are handled in such order that all dependencies are typed and/or evaluated before they are needed. If there is no such an order, there is a cycle, and typing cannot be completed anyway. This order may, however, differ from the order as created by the user, and must preferably be derived before typing and evaluation has started.

An obvious solution is here is to topologically sort the declarations, based on dependencies. A problem is that those dependencies are influenced by the type system, as shown by the overloaded function in our example in the previous paragraph, while this process is applied before type checking begins. Another solution would be, if we encounter a reference to a declaration that has not been typed yet, to suspend the computation for the current declaration, compute the type of the needed declaration first, and then resume typing the first declaration once we have the information required. However, this approach leads to very complex control flows that can easily result in infinite loops. For these reasons, we have decided that, in the current report, we will not handle typing of declarations in the MSOS specification.

References


If you want to receive reports, send an email to: wsinsan@tue.nl (we cannot guarantee the availability of the requested reports).

In this series appeared (from 2009):

09/01 Wil M.P. van der Aalst, Kees M. van Hee, Peter Massuthe, Natalia Sidorova and Jan Martijn van der Werf
Compositional Service Trees

09/02 P.J.J. Cuijpers, F.A.J. Koenders, M.G.P. Pustjens, B.A.G. Senders, P.J.A. van Tilburg, P. Verduin
Queue merge: a Binary Operator for Modeling Queueing Behavior

09/03 Maarten G. Meulen, Frank P.M. Stappers and Tim A.C. Willemse
Breadth-Bounded Model Checking

09/04 Muhammad Atif and MohammadReza Mousavi
Formal Specification and Analysis of Accelerated Heartbeat Protocols

09/05 Michael Franssen
Placeholder Calculus for First-Order logic

09/06 Daniel Trivellato, Fred Spiessens, Nicola Zannone and Sandro Etalle
POLIPO: Policies & OntoLogies for the Interoperability, Portability, and autOnomy

09/07 Marco Zapletal, Wil M.P. van der Aalst, Nick Russell, Philipp Liegl and Hannes Werthner
Pattern-based Analysis of Windows Workflow

09/08 Mike Holenderski, Reinder J. Bril and Johan J. Lukkien
Swift mode changes in memory constrained real-time systems

09/09 Dragan Bošnački, Aad Mathijssen and Yaroslav S. Usenko
Behavioural analysis of an FC Linux Driver

09/10 Ugur Keskin
In-Vehicle Communication Networks: A Literature Survey

09/11 Bas Ploeger
Analysis of ACS using mCRL2

09/12 Wolfgang Boehmer, Christoph Brandt and Jan Friso Groote
Evaluation of a Business Continuity Plan using Process Algebra and Modal Logic

09/13 Luca Aceto, Anna Ingolfsdottir, MohammadReza Mousavi and Michel A. Reniers
A Rule Format for Unit Elements

09/14 Maja Pešić, Dragan Bošnački and Wil M.P. van der Aalst
Enacting Declarative Languages using LTL: Avoiding Errors and Improving Performance

09/15 MohammadReza Mousavi and Emil Sekeriński, Editors
Proceedings of Formal Methods 2009 Doctoral Symposium

09/16 Mohammad Atif
Formal Analysis of Consensus Protocols in Asynchronous Distributed Systems

09/17 Jeroen Keiren and Tim A.C. Willemse
Bisimulation Minimisations for Boolean Equation Systems

09/18 Kees van Hee, Jan Hidders, Geert-Jan Houben, Jan Paredaens, Philippe Thiran
On-the-fly Auditing of Business Processes

10/01 Ammar Osaiweran, Marcel Boosten, MohammadReza Mousavi
Analytical Software Design: Introduction and Industrial Experience Report

10/02 F.E.J. Kruseman Aretz
Design and correctness proof of an emulation of the floating-point operations of the Electrologica X8. A case study
<table>
<thead>
<tr>
<th>Date</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/03</td>
<td>Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, MohammadReza Mousavi and</td>
<td>On Rule Formats for Zero and Unit Elements</td>
</tr>
<tr>
<td></td>
<td>Michel A. Reniers</td>
<td></td>
</tr>
<tr>
<td>10/04</td>
<td>Hamid Reza Asaadi, Rantin Khosravi, MohammadReza Mousavi, Neda Noroozi</td>
<td>Towards Model-Based Testing of Electronic Funds Transfer Systems</td>
</tr>
<tr>
<td></td>
<td>Reinder J. Bril, Uğur Keskin, Moris Behnam, Thomas Nohte</td>
<td>Schedulability analysis of synchronization protocols based on overrun without payback for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hierarchical scheduling frameworks revisited</td>
</tr>
<tr>
<td>10/06</td>
<td>Zvezdan Protić</td>
<td>Locally unique labeling of model elements for state-based model differences</td>
</tr>
<tr>
<td>10/07</td>
<td>C.G.U. Okwudire and R.J. Bril</td>
<td>Converting existing analysis to the EDP resource model</td>
</tr>
<tr>
<td>10/08</td>
<td>Mohammed Atif, Sjoerd Cransen, MohammadReza Mousavi</td>
<td>Reconstruction and verification of group membership protocols</td>
</tr>
<tr>
<td>10/09</td>
<td>Sjoerd Cransen, Jan Friso Groote, Michel Reniers</td>
<td>A linear translation from LTL to the first-order modal μ-calculus</td>
</tr>
<tr>
<td>10/10</td>
<td>Mike Holenderski, Wim Cools Reinder J. Bril, Johan J. Lukkien</td>
<td>Extending an Open-source Real-time Operating System with Hierarchical Scheduling</td>
</tr>
<tr>
<td>10/11</td>
<td>Eric van Wyk and Steffen Zschaler</td>
<td>1st Doctoral Symposium of the International Conference on Software Language Engineering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(SLE)</td>
</tr>
<tr>
<td>10/12</td>
<td>Pre-Proceedings</td>
<td>3rd International Software Language Engineering Conference</td>
</tr>
<tr>
<td>10/13</td>
<td>Faisal Kamiran, Toon Calders and Mykola Pechenizkiy</td>
<td>Discrimination Aware Decision Tree Learning</td>
</tr>
<tr>
<td>10/14</td>
<td>J.F. Groote, T.W.D.M. Kouters and A.A.H. Osaiweran</td>
<td>Specification Guidelines to avoid the State Space Explosion Problem</td>
</tr>
<tr>
<td>10/15</td>
<td>Daniel Trivellato, Nicola Zannone and Sandro Etalle</td>
<td>GEM: a Distributed Goal Evaluation Algorithm for Trust Management</td>
</tr>
<tr>
<td>10/17</td>
<td>L. Aceto, A. Birgisson, A. Ingolfsdottir, and M.R. Mousavi</td>
<td>Decompositional Reasoning about the History of Parallel Processes</td>
</tr>
<tr>
<td>10/18</td>
<td>P.D. Mosses, M.R. Mousavi and M.A. Reniers</td>
<td>Robustness os Behavioral Equivalence on Open Terms</td>
</tr>
<tr>
<td>10/19</td>
<td>Harsh Beohar and Pieter Cuijpers</td>
<td>Desynchronisability of (partial) closed loop systems</td>
</tr>
<tr>
<td>11/01</td>
<td>Kees M. van Hee, Natalia Sidorova and Jan Martijn van der Werf</td>
<td>Refinement of Synchronizable Places with Multi-workflow Nets - Weak termination preserved!</td>
</tr>
<tr>
<td>11/02</td>
<td>M.F. van Amstel, M.G.J. van den Brand and L.J.P. Engelen</td>
<td>Using a DSL and Fine-grained Model Transformations to Explore the boundaries of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Model Verification</td>
</tr>
<tr>
<td>11/03</td>
<td>H.R. Mahrooghi and M.R. Mousavi</td>
<td>Reconciling Operational and Epistemic Approaches to the Formal Analysis of Crypto-Based</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Security Protocols</td>
</tr>
<tr>
<td>11/05</td>
<td>Jan Friso Groote and Jan Lanik</td>
<td>Semantics, bisimulation and congruence results for a general stochastic process operator</td>
</tr>
<tr>
<td>11/06</td>
<td>P.J.L. Cuijpers</td>
<td>Moore-Smith theory for Uniform Spaces through Asymptotic Equivalence</td>
</tr>
<tr>
<td>11/07</td>
<td>F.P.M. Stappers, M.A. Reniers and S. Weber</td>
<td>Transforming SOS Specifications to Linear Processes</td>
</tr>
<tr>
<td>11/08</td>
<td>Debjyoti Bera, Kees M. van Hee, Michiel van Oseh and Jan Martijn van</td>
<td>A Component Framework where Port Compatibility Implies Weak Termination</td>
</tr>
<tr>
<td></td>
<td>der Werf</td>
<td></td>
</tr>
<tr>
<td>11/09</td>
<td>Tsessuren Batsauri, Reinder J. Bril and Johan Lukkien</td>
<td>Model, analysis, and improvements for inter-vehicle communication using one-hop periodic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>broadcasting based on the 802.11p protocol</td>
</tr>
<tr>
<td>Date</td>
<td>Authors/Groups</td>
<td>Title</td>
</tr>
<tr>
<td>-------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>11/10</td>
<td>Neda Noroozi,</td>
<td>Synchronizing Asynchronous Conformance Testing</td>
</tr>
<tr>
<td></td>
<td>Ramtin Khoszavi, MohammadReza Mousavi and Tim A.C. Willemse</td>
<td></td>
</tr>
<tr>
<td>11/11</td>
<td>Jeroen J.A. Keiren and Michel A. Reniers</td>
<td>Type checking mCRL2</td>
</tr>
<tr>
<td>11/12</td>
<td>Muhammad Atif, MohammadReza Mousavi and Ammar Osaiweran</td>
<td>Formal Verification of Unreliable Failure Detectors in Partially Synchronous Systems</td>
</tr>
<tr>
<td>11/13</td>
<td>J.F. Groote, A.A.H. Osaiweran and J.H. Wesselsius</td>
<td>Experience report on developing the Front-end Client unit under the control of formal methods</td>
</tr>
<tr>
<td>11/15</td>
<td>John Businge, Alexander Serebrenik and Mark van den Brand</td>
<td>Eclipse API Usage: The Good and The Bad</td>
</tr>
<tr>
<td>11/17</td>
<td>M.F. van Amstel, A. Serebrenik and M.G.J. van den Brand</td>
<td>Visualizing Traceability in Model Transformation Compositions</td>
</tr>
<tr>
<td>11/18</td>
<td>F.P.M. Stappers, M.A. Reniers, J.F. Groote and S. Weber</td>
<td>Dogfooding the Structural Operational Semantics of mCRL2</td>
</tr>
<tr>
<td>12/01</td>
<td>S. Cranen</td>
<td>Model checking the FlexRay startup phase</td>
</tr>
<tr>
<td>12/02</td>
<td>U. Khadim and P.J.L. Cuijpers</td>
<td>Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP</td>
</tr>
<tr>
<td>12/03</td>
<td>M.M.H.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher</td>
<td>Revised budget allocations for fixed-priority-scheduled periodic resources</td>
</tr>
<tr>
<td>12/04</td>
<td>Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever</td>
<td>Experience Report on Designing and Developing Control Components using Formal Methods</td>
</tr>
<tr>
<td>12/05</td>
<td>Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse</td>
<td>A cure for stuttering parity games</td>
</tr>
<tr>
<td>12/06</td>
<td>A.P. van der Meer</td>
<td>CIF MSOS type system</td>
</tr>
</tbody>
</table>