# Type checking mCRL2

# Type checking mCRL2

Jeroen J.A. Keiren[1] and Michel A. Reniers[2]

[1]Department of Mathematics and Computer Science,
[2] Department of Mechanical Engineering,
Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
`{j.j.a.keiren,m.a.reniers}@tue.nl`

**Abstract**

In this paper we present a type system for the data language of mCRL2, a process algebra based language for formalising the behaviour of communicating system. Much of the type system is standard, and follows the line of, *e.g.*, Pierce [Pie02]. The data language that is described is rich, and supports (infinite) sets and bags, universal and existential quantification, and lambda abstraction. Recursive types can be defined using equational definitions. Subtyping is included for the full data language, and a coercion is given to transform a well-typed expression into a strictly typed expression.

## 1 Introduction

mCRL2 (micro Common Representation Language 2, [GMR+09]) is a language for formalising the behaviour of communicating systems. The language consists of data, processes and logic. The data part of the language is based on higher-order abstract equational data types. The data language is rich, and supports (unbounded) integers and rational numbers, (infinite) sets and bags, lists, structured data types, lambda abstraction and universal and existential quantification. The intention of the data language design is to closely reflect the mathematical counterpart of the data types. The behavioural part of the language is inspired by process algebras, especially ACP, sometimes also referred to as TCP, [BBR09]. It is based on a methodology similar to tools like FDR2 (based on CSP [Hoa85]), CADP [FGK+96] and $\mu$CRL [GP95]. The property specification language of mCRL2 is the modal $\mu$-calculus [Koz83], extended to treat data and time as first class citizens [GW05].

The language mCRL2 is supported by a toolset [mCR]. The toolset provides, among others, an implementation of the data language. In this document we describe a type system for the data language of mCRL2. The intent is to capture the definition of the current type system, but extend and improve upon the current behaviour at these points where the current behaviour is problematic in practice.

Note that the type system as presented in this document is mostly standard, and that the corresponding theory and algorithms can be found in Pierce's seminal book on type checking [Pie02]. We provided references to specific parts of [Pie02] where necessary.

## 2 Preliminaries

We first recall the syntax of types and terms in mCRL2. In mCRL2, types are usually referred to as sort expressions, and terms are referred to as data expressions. In the rest of this paper we use these notions interchangeably.

### 2.1 Types and Terms

The syntax of sort expressions in mCRL2 is defined according to the following grammar:

**Definition 2.1 (Sort expressions)**

$$\begin{array}{lcl} S & ::= & S_{Basic} \mid S \times \cdots \times S \to S \mid \textbf{struct}\ scs, \ldots, scs \mid List(S) \mid Set(S) \mid Bag(S) \\ scs & ::= & f?f \mid f(spj, \ldots, spj)?f \mid f \mid f(spj, \ldots, spj) \\ spj & ::= & f{:}S \mid S \end{array}$$

Here $S_{Basic}$ is a set of basic sorts, that contains at least Bool, Pos, Nat, Int and Real (resp. Booleans, positive numbers, natural numbers, integers and real numbers). In the rest of this paper we write $\mathbb{B}, \mathbb{N}^+, \mathbb{N}, \mathbb{Z}, \mathbb{R}$ for both the syntactic as well as the semantic versions of the numeric types. Functions can have an arbitrary number of arguments, and $\to$ is right-associative. **struct** $scs, \ldots, scs$ describes a "structured sort", where $scs$ are the constructors. A constructor consists of a *name*, a number of arguments ($spj$), and (optionally) the name of a recogniser function. The non-terminal $spj$ describes a constructor argument, which consists of an optional name and a sort. Names are denoted by $f$, and the ? seperates the constructor from its recogniser.

A subtype ordering on the (standard) numeric data types is also present, such that $\mathbb{N}^+ <: \mathbb{N} <: \mathbb{Z} <: \mathbb{R}$. This relation is lifted to the other type structures. It is not possible for the user to add his own subtype relations.

Before we give the definition of a data expression, we first illustrate the use of structured sorts with an example.

**Example 2.2** The following structured sort defines a type in which each expression is either *nil*, or a pair of natural numbers.

$$\textbf{struct}\ pair(left{:}\mathbb{N}, right{:}\mathbb{N})?is\_pair \mid nil?is\_nil$$

This expression is a sort expression, meaning there can be expressions of sort **struct** $pair(left{:}\mathbb{N}, right{:}\mathbb{N})?is$ $nil?is\_nil$. Examples of elements of this sort are $pair(0, 1)$ and $nil$.

A structured sort is not just the name of a sort; it also specifies its structure, and some functions to manipulate and query expressions of the sort. For the structured sort given above, the following functions generated:

$$\begin{array}{rl} left: & \textbf{struct}\ pair(left{:}\mathbb{N}, right{:}\mathbb{N})?is\_pair \mid nil?is\_nil \to \mathbb{N} \\ right: & \textbf{struct}\ pair(left{:}\mathbb{N}, right{:}\mathbb{N})?is\_pair \mid nil?is\_nil \to \mathbb{N} \\ is\_pair: & \textbf{struct}\ pair(left{:}\mathbb{N}, right{:}\mathbb{N})?is\_pair \mid nil?is\_nil \to \mathbb{B} \\ is\_nil: & \textbf{struct}\ pair(left{:}\mathbb{N}, right{:}\mathbb{N})?is\_pair \mid nil?is\_nil \to \mathbb{B} \end{array}$$

Applied to an expression of the form $pair(x, y)$ the function *left* retrieves the first argument of the pair, *right* retrieves the second argument of the pair; applied to any other argument the function is undefined. The function *is_pair* applied to an expression of the form $pair(x, y)$ returns true, and false for expressions of any other form. The definition of *is_nil* is similar.

Data expressions adhere to the following syntax.

**Definition 2.3 (Data expressions)** We inductively define data expressions $e$, with sort expressions $S$ as follows.

$$\begin{array}{rcl} e & ::= & x \mid f \mid e(e,\ldots,e) \mid \lambda\vec{x}{:}\vec{S}.e \mid \forall\vec{x}{:}\vec{S}.e \mid \exists\vec{x}{:}\vec{S}.e \\ & \mid & e \textbf{ whr } \vec{x} = \vec{e} \textbf{ end} \mid \{x{:}S \mid e\} \end{array}$$

Here $x$ represents a variable, and $f$ represents a function symbol. We write $\vec{x}{:}\vec{S}$ to denote a vector of the form $x{:}S,\ldots,x{:}S$, $\vec{x} = \vec{e}$ to denote a vector $x = e,\ldots,x = e$, and we write $\vec{v}_i$ to denote the $i$-th element of such a vector. The data expression $e(e,\ldots,e)$ denotes the application of a data expression to some others, $\lambda\vec{x}{:}\vec{S}.e$ denotes lambda abstraction. $\forall\vec{x}{:}\vec{S}.e$ and $\exists\vec{x}{:}\vec{S}.e$ describe universal and existential quantification, respectively. Set and bag comprehension are denoted by $\{x{:}S \mid e\}$, where the actual type depends on the sort of $e$. It is a set if $e$ is Boolean and a bag if $e$ is a natural number.

In our exposition we typically use $x, y, z$ for variables and $f, g, h$ as function symbols; for operations defined on standard data types we sometimes use infix notation instead of prefix notation. We for example write $2 + 3$ instead of $+(2, 3)$. For types we typically use $S, T$.

## 2.2 Data specification

In mCRL2 all functions that are used have to be declared. For the standard data types standard definitions of a large number of functions have been provided, along with an efficient implementation. For a detailed account of all standard data types we refer to [GR10].

The user can define data types in a specification consisting of three parts: (1) sort declarations, where the sorts are defined; (2) function declarations, where the functions operating on the sorts from (1) are declared (note that they may also operate on system defined sorts); (3) an equational specification, in which the functions are defined by means of (guarded) equations.

Note that usually two classes of function declarations are distinguished in mCRL2, namely the constructors and the mappings. The constructors inductively describe the elements of a sort, whereas the mappings can be arbitrary functions over declared sorts. Because the constructor functions describe the elements of a data type, allowing widening of types in constructor functions is undesired. Note that the standard data types, which are the only basic data types for which we allow subtyping, incrementally extend each other. Hence, this is not problematic in practice. For the purpose of this paper, we therefore do not need to distinguish constructors and mappings, hence we just refer to them collectively as functions.

Sort declarations occur in two forms. Either they just declare a basic sort (*i.e.*, it just declares a *name* of a sort), or they declare two sorts to be equal, called *type aliases.*

**Example 2.4** The following specification declares three sorts. The first is a basic sort with the name *Colour*, the second is a sort *Tree*, describing to the sort of binary trees, and the third is a sort *Address*, which is defined to be equal to the natural numbers (*i.e.*, *Address* is a type alias for $\mathbb{N}$).

$$\begin{array}{llcl} \textbf{sort} & \mathit{Colour}; & & \\ & \mathit{Tree} & = & \textbf{struct } \mathit{node}(\mathit{op} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}, \mathit{left} : \mathit{Tree}, \mathit{right} : \mathit{Tree}) \mid \mathit{leaf}(b : \mathbb{B}); \\ & \mathit{Address} & = & \mathbb{N}; \end{array}$$

Function declarations are used to declare the signature of a function. They have the form $f{:}S$, where $f$ is a name, and $S$ is a sort expression. Note that overloading of function symbols is allowed, and is in fact heavily used by the definitions of the standard data types, as is illustrated by the following example.

**Example 2.5** Consider addition of numbers. The following are the function declarations for addition (+) of numbers, as defined in the standard data types

$$\begin{array}{l}
+{:}\mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}^+ \\
+{:}\mathbb{N} \times \mathbb{N}^+ \to \mathbb{N}^+ \\
+{:}\mathbb{N}^+ \times \mathbb{N} \to \mathbb{N}^+ \\
+{:}\mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
+{:}\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
+{:}\mathbb{R} \times \mathbb{R} \to \mathbb{R}
\end{array}$$

The definition of data types is provided by an equational specification, with equations of the form $c \to t_1 = t_2$, meaning that, if Boolean condition $c$ holds, $t_1$ and $t_2$ are equal. Note that the types of $t_1$ and $t_2$ must be the same. In the mCRL2 toolset these rules are interpreted in a left-to-right fashion, allowing the reduction of expressions to a normal form using term rewriting [Wee07].

**Example 2.6** Consider the Booleans, with constructors $\mathsf{true}$ and $\mathsf{false}$, and mapping $\wedge{:}\mathbb{B} \times \mathbb{B} \to \mathbb{B}$, and variable $b{:}\mathbb{B}$. Conjunction ($\wedge$) is characterised using the following equations.

$$\begin{array}{lcl}
b \wedge \mathsf{true} & = & b \\
b \wedge \mathsf{false} & = & \mathsf{false} \\
\mathsf{true} \wedge b & = & b \\
\mathsf{false} \wedge b & = & \mathsf{false}
\end{array}$$

In the rest of this paper we use the following notation. $S_{Basic}$ is the set of basic sorts (including $\mathbb{B}, \mathbb{N}^+, \mathbb{N}, \mathbb{Z}, \mathbb{R}$), $E$ is a set of type aliases, and $\Omega$ is a set of function declarations. For the purpose of type checking, the equational specification of data types is irrelevant. We combine the declarations into the structure $\Sigma = (S_{Basic}, E, \Omega)$, to which we also refer as signature. Throughout the rest of this paper we assume a given signature $\Sigma = (S_{Basic}, E, \Omega)$.

## 2.3 Fixed point basics

Our algorithms for subtyping are based on fixed point iteration. We therefore first recall some basic fixed point theory.

In this section we assume some fixed universe $U$. A function $F \in 2^U \to 2^U$ is monotone if $X \subseteq Y \implies F(X) \subseteq F(Y)$ for all $X, Y \subseteq U$. For subsets $X$ of $U$ we say that $X$ is (1) $F$-closed if $F(X) \subseteq X$, (2) $F$-consistent if $X \subseteq F(X)$, and (3) a fixed point of $F$ if $F(X) = X$.

**Theorem 2.7** [Knaster-Tarski [Tar55]] Let $F$ be a monotone function.

- The intersection of all $F$-closed sets is the least fixed point of $F$, denoted $\mu F$, and
- the union of all $F$-consistent sets is the greatest fixed point of $F$, denoted $\nu F$.

In this paper we want to check for some element $x$ whether it is in the fixed point of a function $F$. In general, this computation can cause an exponential blow-up. However, for the set of *invertible functions*, the computation can be done more efficiently.

**Definition 2.8** Consider a function $F$ and a set $U$, and let $G_x$ be the following collection of sets.

$$G_x = \{X \subseteq U \mid x \in F(X)\}$$

Function $F$ is *invertible* if for all $x \in U$ either $G_x$ is empty, or there is a *unique* element $X \in G_x$ such that $\forall Y \in G_x : X \subseteq Y$, *i.e.* there is a unique element in $G_x$ that is a subset of all the others.

When a function is invertible, we can define a *support function* as follows:

**Definition 2.9** Let $F$ be an invertible function. The partial function $\mathit{supp}_F{:}U \rightharpoonup 2^U$ is defined by

$$\mathit{supp}_F(x) = \begin{cases} X & \text{if } X \in G_x \text{ and } \forall Y \in G_x : X \subseteq Y \\ \bot & \text{otherwise} \end{cases}$$

The support function is lifted to sets as follows.

$$\mathit{supp}_F(X) = \begin{cases} \bigcup_{x \in X} \mathit{supp}_F(x) & \text{if } \forall x \in X : \mathit{supp}_F(x) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Using a support set, algorithms have been developed for checking membership in the least and greatest fixed points of a generating function $F$ [Pie02, Section 21.5]. The algorithms involve asking how $x$ could have been generated by $F$. Having an invertible $F$ ensures that a given $x$ can only be generated in a single way. This prevents the combinatorial explosion one is faced with for a non-invertible $F$.

## 3 Basic type checking

We first introduce plain typechecking of mCRL2. In the following sections we extend the type checking with subtyping, and we add recursive types. Observe that the data language in mCRL2 is closely related to the simply typed lambda-calculus. It therefore may come as no surprise that the type system of mCRL2 is similar to the type system for simply typed lambda calculus [Pie02, Chapter 9].

Well-typedness of data expressions, expressed using statements of the form $\Gamma \vdash_\Sigma e : S$, is defined using a number of syntax-directed derivation rules. In this inference system we use signature $\Sigma$, as well as a context $\Gamma$ which records our variable declarations. Note that $\Gamma$ operates as a stack, and $\Gamma, x{:}S$ denotes the extension of $\Gamma$ with a declaration of variable $x$ of type $S$. With $x{:}S \in \Gamma$ we denote that the topmost declaration of the variable with name $x$ in $\Gamma$ has type $S$, *i.e.*, $x{:}T \notin \Gamma, x{:}S$, for $S \neq T$, regardless of the declarations in $\Gamma$. Extension of $\Gamma$ with a vector $\vec{x}{:}\vec{S}$ is an abbreviation for $\Gamma, x_1{:}S_1, \ldots, x_n{:}S_n$ for $\vec{x}{:}\vec{S} = x_1{:}S_1, \ldots, x_n{:}S_n$ We use $\Gamma$ and $\Sigma$ to define well-typedness of data expressions as follows.

$$\frac{x : S \in \Gamma}{\Gamma \vdash_\Sigma x : S}\text{(T-Var)} \qquad \frac{f : S_1 \times \cdots \times S_n \to T \in \Sigma}{\Gamma \vdash_\Sigma f : S_1 \times \cdots \times S_n \to T}\text{(T-Func)}$$

The rules T-Var and T-Func are basic rules checking whether a variable or a function has been declared. T-Var looks up the declaration of a variable in the context, whereas T-Func looks up the declaration of a function in the signature.

$$\frac{\Gamma, \vec{x}{:}\vec{S} \vdash_\Sigma e : T}{\Gamma \vdash_\Sigma \lambda\vec{x}{:}\vec{S}.e : S_1 \times \cdots \times S_n \to T} \text{(T-Abs)}$$

T-Abs determines the type of a lambda abstraction. Note that this rule extends the context with the variables bound by the abstraction, and that the result is a function type.

$$\frac{\Gamma \vdash_\Sigma e{:}S_1 \times \cdots \times S_n \to T \quad \Gamma \vdash_\Sigma e_1{:}S_1 \quad \cdots \quad \Gamma \vdash_\Sigma e_n{:}S_n}{\Gamma \vdash_\Sigma e(e_1, \ldots, e_n) : T} \text{(T-Appl)}$$

T-Appl denotes application of a function to a number of arguments. Observe that the number of arguments must coincide with the arity of the head of the application, and that the head of the application may be an arbitrary term with a function type. The types of the arguments and the function must coincide.

$$\frac{\Gamma \vdash_\Sigma d_1 : S_1 \quad \cdots \quad \Gamma \vdash_\Sigma d_n : S_n \quad \Gamma, \vec{x}{:}\vec{S} \vdash_\Sigma e : T}{\Gamma \vdash_\Sigma e \ \mathbf{whr}\ \vec{x} = \vec{d}\ \mathbf{end} : T} \text{(T-Where)}$$

The rule for where-clauses is interesting in that this is the only place where the type of a variable is determined from the type of an expression, instead of having it declared by the user. For where-clauses the restriction applies that for all $i, j$ variable $x_i$ may not occur in expression $d_j$. As a result, only the body $e$ of the clause needs to be checked in an extended context, where the types of the declared variables $x_i$ are inferred from the expressions $e_i$.

$$\frac{\Gamma, \vec{x}{:}\vec{S} \vdash_\Sigma e : \mathbb{B}}{\Gamma \vdash_\Sigma \forall\vec{x}{:}\vec{S}.e : \mathbb{B}} \text{(T-Forall)} \qquad \frac{\Gamma, \vec{x}{:}\vec{S} \vdash_\Sigma e : \mathbb{B}}{\Gamma \vdash_\Sigma \exists\vec{x}{:}\vec{S}.e : \mathbb{B}} \text{(T-Exists)}$$

The types for universal and existential quantification are inferred much like the type for lambda abstraction, only in this case the result, and the body of the expression, are required to be Boolean.

$$\frac{\Gamma, x : S \vdash_\Sigma e : \mathbb{B}}{\Gamma \vdash_\Sigma \{x : S \mid e\} : \mathit{Set}(S)} \text{(T-Set)} \qquad \frac{\Gamma, x : S \vdash_\Sigma e : \mathbb{N}}{\Gamma \vdash_\Sigma \{x : S \mid e\} : \mathit{Bag}(S)} \text{(T-Bag)}$$

Finally we allow for the definition of set and bag comprehension. The body of a set comprehension is a predicate defining the elements that are part of the set. For bag comprehension the number of times each element occurs in the bag is defined. Note that, contrary to lambda abstraction and universal and existential quantification, set and bag abstraction only allow binding of a single variable. The reason for disallowing multiple variable binding is the way set and bag comprehension is interpreted in mCRL2. The expression $\{x{:}S \mid e\}$, in which $e$ is Boolean, describes the set containing the elements of type $S$ satisfying predicate $e$, where $e$ may use $x$. In this notation, $x{:}S$ is the bound variable, and $x$'s of type $S$ are the elements that get collected, as long as they satisfy $e$. In short, $x$ serves the role of a bound variable, and it provides a characterisation of elements that are collected. A similar rationale holds for bags.

In computer science, sometimes an alternative notation for set comprehension is advocated, see *e.g.* [NK04, Section 10.2]. Instead of $\{x{:}S \mid e\}$, the same set is written as

$\{x{:}S \mid e \mid x\}$, where the first part gives the bound variables, the second part gives a predicate, and the third part describes the elements that are collected. In this case, we can collect other elements than $x$'s, for example $x^2$s:

$$\{x{:}\mathbb{Z} \mid -3 \leq x \leq 100 \mid x^2\}$$

The above describes the set consisting of the squares of all integers between $-3$ and 100 (inclusive). The extension of bags to this notation is not clear. Consider for example the following bag:

$$\{n{:}\mathbb{N} \mid \mathsf{true} \mid 0\}$$

This bag would contain 0 infinitely many times.

As a result of the similar form of the expressions for set and bag comprehension, and overloading that can take place, it can occur that an expression has both types $Set(S)$ and $Bag(S)$ for some sort $S$.

**Example 3.1** Let $f{:}S \to \mathbb{B}, f{:}S \to \mathbb{N} \in \Sigma$, then there are valid derivations for $\{x{:}S \mid f(x)\}{:}Set(S)$ and $\{x{:}S \mid f(x)\}{:}Bag(S)$.

## 4 Subtyping

The standard data types of mCRL2 include several numeric data types, *viz.* $\mathbb{N}^+$, $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$. From the point of view of user-friendliness it is desirable to include a subtyping relation for these types, as failing to do so would require the user to make all casts explicit, as illustrated in the following example.

**Example 4.1** Suppose $f{:}\mathbb{Z} \to \mathbb{B}$ is a function expecting an integer argument, and $x{:}\mathbb{N}$ is a natural number. If no subtyping mechanism is included in the type system, the following expression is ill-typed.

$$f(x)$$

The language would have to provide some function $N2I{:}\mathbb{N} \to \mathbb{Z}$, explicitly casting natural numbers to integers, requiring the user to write the following.

$$f(N2I(x))$$

**Example 4.2** As a more concrete example of the desire for a subtyping mechanism, consider the following expression:

$$0 = 1$$

Where 0 is of type $\mathbb{N}$, and 1 is of type $\mathbb{N}^+$, and we want to establish whether 0 and 1 are equal. Without subtyping mechanism, the expression is ill-typed, as $=$ expects arguments of equal type, hence the user instead has to write

$$0 = N2I(1)$$

## 4.1 Basic subtyping

Writing explicit casts is cumbersome and error-prone, hence we introduce a subtyping relation $<:$, where $S <: T$ denotes that $S$ is a *subtype of $T$*. The first three rules are the basic axioms describing the hierarchy of numeric data types. Note that these are the only basic sorts for which $<:$ holds asymmetrically, *i.e.*, we do not allow the user to introduce subtype relations. The rest of our system for subtyping is again standard, and an excellent exposition of subtyping can be found in [Pie02, Chapter 15]. In that same chapter it also shown how to perform coercion, *i.e.*, how to automatically add casts to the code being typechecked in order to obtain a *strictly typed* expression. A type assertion is called *strict* if it can be proven without using T-Sub.

$$\frac{}{\mathbb{N}^+ <: \mathbb{N}}\ \text{(S-P2N)} \qquad \frac{}{\mathbb{N} <: \mathbb{Z}}\ \text{(S-N2I)} \qquad \frac{}{\mathbb{Z} <: \mathbb{R}}\ \text{(S-I2R)}$$

Including the following rules, which provide the transitive closure of the previous axioms, alleviates the need for a rule for transitivity.

$$\frac{}{\mathbb{N}^+ <: \mathbb{Z}}\ \text{(S-P2I)} \qquad \frac{}{\mathbb{N}^+ <: \mathbb{R}}\ \text{(S-P2R)} \qquad \frac{}{\mathbb{N} <: \mathbb{R}}\ \text{(S-N2R)}$$

The following rule provides reflexivity:

$$\frac{}{S <: S}\ \text{(S-Refl)}$$

In our example derivations we usually omit applications of this rule.

The definition of $<:$ extends naturally to container types and structured sorts.

$$\frac{S <: T}{\mathit{List}(S) <: \mathit{List}(T)}\ \text{(S-List)} \qquad \frac{S <: T}{\mathit{Set}(S) <: \mathit{Set}(T)}\ \text{(S-Set)} \qquad \frac{S <: T}{\mathit{Bag}(S) <: \mathit{Bag}(T)}\ \text{(S-Bag)}$$

The subtyping relation for function types is covariant on the codomain, and contravariant on the domain.

$$\frac{T_1 <: S_1 \quad \cdots \quad T_n <: S_n \quad S <: T}{S_1 \times \cdots \times S_n \to S <: T_1 \times \cdots \times T_n \to T}\ \text{(S-Func)}$$

We also allow subtyping of structured sorts. The names of recognisers and arguments need to be syntactically equal. Note that we omit the names of recognisers and arguments here for the sake of brevity.

$$\frac{S_{i,j} <: T_{i,j} \text{ for all } i, j}{\mathit{Struct1} <: \mathit{Struct2}}\ \text{(S-Struct)}$$

where *Struct1* is defined as

$$\begin{array}{l}\mathbf{struct} f_1(S_{1,1}, \ldots, S_{1,m_1})\\ \quad\vdots\\ \quad | f_n(S_{n,1}, \ldots, S_{n,m_n})\end{array}$$

and *Struct2* is defined as

$$\begin{array}{l}\mathbf{struct} f_1(T_{1,1}, \ldots, T_{1,m_1})\\ \quad\vdots\\ \quad | f_n(T_{n,1}, \ldots, T_{n,m_n})\end{array}$$

Extending the rules for typing data expressions with the following rule adds all features of subtyping to our inference system.

$$\frac{\Gamma \vdash_\Sigma e : T \quad \Gamma \vdash_\Sigma T <: S}{\Gamma \vdash_\Sigma e : S}\ (\text{T-Sub})$$

The rule for subtyping of function types may seem strange at a first glance. In subtyping of functions one function type is a subtype of another if one of its domain types is larger than the corresponding domain type in the other function type. The reasons for this are illustrated by the following example.

**Example 4.3** Let $\mathit{map}{:}(\mathbb{Z} \to S) \times \mathit{List}(\mathbb{Z}) \to \mathit{List}(S)$, and $f{:}\mathbb{N} \to S$ be functions, and let $x{:}\mathit{List}(\mathbb{Z})$ be a variable. Suppose we use the following alternative rule for subtyping.

$$\frac{S_1 <: T_1 \quad \cdots \quad S_n <: T_n \quad S <: T}{S_1 \times \cdots \times S_n \to S <: T_1 \times \cdots \times T_n \to T}\ (\text{S-ReverseFunc})$$

This gives rise to the following type derivation for the expression $\mathit{map}(f, x)$.

$$\frac{\dfrac{\mathit{map}{:}(\mathbb{Z} \to S) \times \mathit{List}(\mathbb{Z}) \to \mathit{List}(S) \in \Sigma}{\Gamma \vdash_\Sigma \mathit{map}{:}(\mathbb{Z} \to S) \times \mathit{List}(\mathbb{Z}) \to \mathit{List}(S)}\ (\star) \qquad \dfrac{(1)}{\Gamma \vdash_\Sigma f{:}\mathbb{Z} \to S} \qquad \dfrac{x{:}\mathit{List}(\mathbb{Z}) \in \Gamma}{\Gamma \vdash_\Sigma x{:}\mathit{List}(\mathbb{Z})}\ (\text{T-Var})}{\Gamma \vdash_\Sigma \mathit{map}(f, x){:}\mathit{List}(S)}\ (\text{T-Appl})$$

Where at $(\star)$ we have applied T-Func, and subderivation (1) is defined as follows:

$$\frac{\dfrac{f{:}\mathbb{N} \to S \in \Gamma}{\Gamma \vdash_\Sigma f{:}\mathbb{N} \to S}\ (\text{T-Func}) \qquad \dfrac{\dfrac{}{\mathbb{N} <: \mathbb{Z}}\ (\text{S-N2I})}{\mathbb{N} \to S <: \mathbb{Z} \to S}\ (\text{S-ReverseFunc})}{\Gamma \vdash_\Sigma f{:}\mathbb{Z} \to S}\ (\text{T-Sub}) \tag{1}$$

We see that given our alternative rule, we find a valid type derivation in this case. If we take a closer look at our expression, and we give *map* the classical meaning of applying a function to every element of a list, we conclude that this expression should be regarded as invalid. Our derivation allows the application of a function which only accepts natural numbers to elements of integer type, whereas for negative values the function is undefined. Using the rule S-Func we are not be able to infer a type for expression $\mathit{map}(f, x)$, as desired.

The intuition behind the rule for subtyping function types is that it allows functions to be more widely applicable, and to return values from a constrained set.

Observe that in the definition for subtyping, the rules for $\mathit{Set}(S)$ and functions of $S \to \mathbb{B}$ and $\mathit{Bag}(S)$ and functions of $S \to \mathbb{N}$ are not isomorphic. As sets and bags are predefined-defined, we know for each widening how to extend the definition. This is illustrated by the following example.

**Example 4.4** Consider a set of natural numbers $s : \mathit{Set}(\mathbb{N})$. If we widen this to $s : \mathit{Set}(\mathbb{Z})$, then we know that for each negative number $n$, $n$ is not in $s$. If we consider an arbitrary function $f : \mathbb{N} \to \mathbb{B}$ however, we do not know how to extend the definition of $f$ to negative numbers automatically, in order to obtain $f : \mathbb{Z} \to \mathbb{B}$.

Similarly, for a bag of integers $b : \mathit{Bag}(\mathbb{Z})$, if we widen this to $b : \mathit{Bag}(\mathbb{R})$, then we know for each number $r$, which is not in $\mathbb{Z}$ that it is in $b$ zero times.

In order to not overly complicate the data language, a conscious decision was made to forbid width-subtyping of structured sorts and the existing numeric types, like in the following example.

**Example 4.5** Forms of subtyping that are disallowed by convention.

$$\begin{aligned} \textbf{struct } \mathit{one} &\not<: \textbf{struct } \mathit{one} \mid \mathit{two} \\ \{0, \ldots, 5\} &\not<: \mathbb{N} \end{aligned}$$

where $\{0, \ldots, 5\}$ means the type consisting of natural numbers $0, \ldots, 5$.

**Property 4.6** The subtype relation is a preorder.

**Remark 4.7** An algorithm for subtyping can be constructed based on the inference rules. To obtain such an algorithm, we want to have a set of syntax directed deduction rules. We obtain such a system by removing the rule T-Sub, and replacing the rule T-Appl by the following rule TA-Appl. Note that removing S-Refl is straightforward.

$$\frac{\begin{array}{ccc} & \Gamma \vdash_\Sigma e_1{:}S_1 & S_1 <: T_1 \\ & \vdots & \vdots \\ \Gamma \vdash_\Sigma e{:}T_1 \times \cdots \times T_n \to S & \Gamma \vdash_\Sigma e_n{:}S_n & S_n <: T_n \end{array}}{\Gamma \vdash_\Sigma e(e_1, \ldots, e_n) : S} \text{ (TA-Appl)}$$

## 4.2 Fixpoint characterisation of subtyping

In the next section, we are going to extend the type system with recursive types. For subtyping in the context of recursive types, a fixed-point characterisation of the subtyping relation is convenient. To make a swift transition to recursive types, we first introduce the fixed point characterisation of subtyping in the simpler setting of this section. For a description of the fixpoint characterisation introduced here, see [Pie02, Section 21.8].

**Trees of types** The set of finite tree types $\mathcal{T}_f$ is the least fixed point of the generating function described by the grammar of types. The universe of this generating function is the set of all finite and infinite trees labelled with Unit, $S_{Basic}$, $\to$, *List*, *Set*, *Bag*, **struct**, and $\times$, where $\times$ is merely used to separate the arguments of constructors of structured sorts. We also label the edges with an index in the domain of $\to$, $\mathcal{C}$ for the codomain, the name of a constructor in case of **struct**, and $\mathcal{A}$ for the element type of a container.

Note that here we have introduced Unit to denote constructors of structured sorts that have no arguments.

**Example 4.8** The tree corresponding to the type $S_1 \times S_2 \to \textbf{struct } \mathit{leaf} \mid \mathit{complex}(\mathit{List}(S))$ is the following:

$$\begin{tikzcd} & \to \arrow[dl, "1"'] \arrow[d, "2"] \arrow[dr, "\mathcal{C}"] & & \\ S_1 & S_2 & \textbf{struct} \arrow[dl, "\text{leaf}"'] \arrow[dr, "\text{complex}"] & \\ & \mathsf{Unit} & & \mathit{List} \arrow[d, "\mathcal{A}"] \\ & & & S \end{tikzcd}$$

**Finite subtyping using fixed points** Two finite tree types $S$ and $T$ are in the subtype relation if $(S,T) \in \mu S_f$, where $S_f$ is the monotone function $S_f \in 2^{\mathcal{T}_f \times \mathcal{T}_f} \to 2^{\mathcal{T}_f \times \mathcal{T}_f}$, defined by

$$\begin{array}{rcl}
S_f(R) & = & \{(\mathbb{N}^+,\mathbb{N}),(\mathbb{N}^+,\mathbb{Z}),(\mathbb{N}^+,\mathbb{R}),(\mathbb{N},\mathbb{Z}),(\mathbb{N},\mathbb{R}),(\mathbb{Z},\mathbb{R})\} \\
 & \cup & \{(\mathcal{C}(s),\mathcal{C}(t)) \mid (s,t) \in R, \mathcal{C} \in \{\mathit{List},\mathit{Set},\mathit{Bag}\}\} \\
 & \cup & \{(S_1 \times \cdots \times S_n \to S, T_1 \times \cdots \times T_n \to T) \mid (T_i,S_i),(S,T) \in R, 1 \le i \le n\} \\
 & \cup & \left\{ \left( \begin{array}{cc} \mathbf{struct} f_1(S_{1,1},\ldots,S_{1,m_1}) & \mathbf{struct} f_1(T_{1,1},\ldots,T_{1,m_1}) \\ \vdots & \vdots \\ |f_n(S_{n,1},\ldots,S_{n,m_n}), & |f_n(T_{n,1},\ldots,T_{n,m_n}) \end{array} \right) \middle| \begin{array}{r} 1 \le i \le n \\ 1 \le j \le m_i \\ (S_{i,j},T_{i,j}) \in R \end{array} \right\}
\end{array}$$

Intuitively, $S <: T$ can be derived in the inference system if and only if $(S,T)$ is in the least fixed point of $S_f$ applied to the identity relation.

**Lemma 4.9** $\Gamma \vdash_\Sigma S <: T$ if and only if $(S,T) \in \mu S_f(\mathcal{I})$, where $\mathcal{I}$ is the identity relation.

A fixed point algorithm can be used to determine whether $S <: T$. For this we establish some properties of $S_f$. We first define the support set of $S_f$.

$$\mathit{supp}_{S_f}(S,T) = \begin{cases}
\emptyset & \text{if } (S,T) \in \{(\mathbb{N}^+,\mathbb{N}),(\mathbb{N}^+,\mathbb{Z}), \\
 & (\mathbb{N}^+,\mathbb{R}),(\mathbb{N},\mathbb{Z}),(\mathbb{N},\mathbb{R}),(\mathbb{Z},\mathbb{R})\} \\
\{(S,T)\} & \text{if } S = \mathcal{C}(S_1) \text{ and } T = \mathcal{C}(T_1) \\
 & \text{for } \mathcal{C} \in \{\mathit{List},\mathit{Set},\mathit{Bag}\} \\
\{(S',T')\} \cup \{1 \le i \le n \mid (T_i,S_i)\} & \text{if } S = S_1 \times \cdots \times S_n \to S' \\
 & \text{and } T = T_1 \times \cdots \times T_n \to T' \\
\{1 \le i \le n, 1 \le j \le m_i \mid (S_{i,j},T_{i,j})\} & \text{if } S = \mathbf{struct} f_1(S_{1,1},\ldots,S_{1,m_1}) \\
 & \qquad \vdots \\
 & \qquad |f_n(S_{n,1},\ldots,S_{n,m_n}) \\
 & T = \mathbf{struct} f_1(T_{1,1},\ldots,T_{1,m_1}) \\
 & \qquad \vdots \\
 & \qquad |f_n(T_{n,1},\ldots,T_{n,m_n}) \\
\bot & \text{otherwise}
\end{cases}$$

Intuitively, the support function can be used, given some pair $(S,T)$, to uniquely determine the next step needed in the computation, in order to establish whether $S <: T$.

**Lemma 4.10** The function $S_f$ is invertible.

Because $S_f$ is invertible, $S <: T$ can be checked efficiently by determining whether $(S,T) \in \mu S_f(\mathcal{I})$, *i.e.*, the pair $(S,T)$ is in the least fixed point of $S_f$ applied to the identity relation.

Note that for the setting with finite types, using a fixed point algorithm to check the subtyping relation is overkill. We merely describe it here to extend it to infinite types in the next section.

## 5 Recursive types

In mCRL2 the user can define infinite types using recursive definitions. An example of such an infinite type is the following.

**Example 5.1** A binary tree with natural numbers as leafs can be defined as follows.

$$\mathit{Tree} = \textbf{struct}\ \mathit{node}(\mathit{Tree}, \mathit{Tree}) \mid \mathit{leaf}(\mathbb{N})$$

### 5.1 Normalisation

To cope with type aliases in an implementation, we use a rewrite system, rewriting each type in the system to a normal form. Observe that the rules $S = T$ in the definition of aliases (say $E$) can come in the following different forms:

$$\begin{array}{rcl}
S & = & T \text{ with } T \in S_{Basic} \\
S & = & T_1 \times \cdots \times T_n \to T \\
S & = & \mathcal{C}(T) \text{ with } \mathcal{C} \in \{\mathit{List}, \mathit{Set}, \mathit{Bag}\} \\
S & = & \textbf{struct} f_1(S_{1,1}, \ldots, S_{1,m_1}) \\
 & & \qquad\vdots \\
 & & \qquad |f_n(S_{n,1}, \ldots, S_{n,m_n})
\end{array}$$

We interpret the equivalences as follows. We order some rules left-to-right, and others right-to-left in a rewrite system $\mathcal{R}$, where the directions are given according to the following rules.

$$\begin{array}{ll}
S \to T & \text{if } S = T \in E \text{ and } T \in S_{Basic} \\
S \to \mathcal{C}(T) & \text{if } S = \mathcal{C}(T) \in E \text{ and } \mathcal{C} \in \{\mathit{List}, \mathit{Set}, \mathit{Bag}\} \\
\textbf{struct} f_1(S_{1,1}, \ldots, S_{1,m_1}) \ \to S & \text{if } S = \ \textbf{struct} f_1(S_{1,1}, \ldots, S_{1,m_1}) \ \in E \\
\qquad\vdots & \qquad\qquad\vdots \\
\qquad |f_n(S_{n,1}, \ldots, S_{n,m_n}) & \qquad\qquad |f_n(S_{n,1}, \ldots, S_{n,m_n}) \\
S \to T_1 \times \cdots \times T_n \to T & \text{if } S = T_1 \times \cdots \times T_n \to T \in E
\end{array}$$

In principle, the rules are ordered such that we preserve as much structure as possible, *i.e.* from the structure of a type, we can still infer what kind of expression we are dealing with. The only exception to this rule lies in structured sorts. We fold structured sorts, as their structure is too large to include in terms for any practical applications, and especially, repeated unfolding does not terminate, and does not yield unique fixed points. We pose some restrictions on sort aliases to ensure termination of normalisation using $\mathcal{R}$.

#### Restrictions on sort aliases

- Recursions in right hand sides of all aliases, except for structured sorts, must be loop free, *i.e.*, a sort, except for a structured sort, cannot recursively depend on itself (both directly and indirectly). As an example, $S = T \to U, U = S$ is not allowed, as $S$ occurs in the right hand side of $U$, which occurs in the right hand sided of $S$, and hence a loop is formed.

- Sorts can occur at the left hand side of at most one sort equivalence in $E$, *i.e.* left hand sides of aliases are unique.
- Only basic sorts, other than predefined sorts can occur at left-hand sides of equivalences in $E$.

**Lemma 5.2** A rewrite system that satisfies the above restrictions on $E$ is terminating.

**Proof sketch**

- Rewriting a structured sort terminates, as it is rewritten from right-to-left, and its left hand side may occur as left hand side of only one equation. Applying this rule hence decreases the number of structured sorts.
- Observe that the number of rewrite rules is finite, and no sort depends recursively on itself, hence in every sequence of rewrite steps, a single rule is never applied twice to the same expression, and rewriting terminates.

$\square$

In order to obtain a strongly normalising, and confluent rewrite system, we apply Knuth-Bendix completion on the rewrite system. If we have a rule $f(g(t)) \to u_1$, and $g(t) \to u_2$, *i.e.*, the left hand side of one rewrite rule is a subterm of the left hand side of another rewrite rule, than we add the rewrite rule $f(u_2) \to \mathcal{R}(u_1)$, where $\mathcal{R}(u_1)$ is the normal form of $u_1$ with respect to the current rewrite system.

**Lemma 5.3** Given the above restrictions on the equations in $E$, Knuth-Bendix completion terminates, the resulting rewrite system is strongly normalising and confluent.

## 5.2 Subtyping

Having recursion through the definition of aliases introduces the problem of subtyping in the context of infinite types. Observe that in the syntax of mCRL2, defining the name of the type (*Tree*), and defining the type itself are mixed. For our exposition on subtyping, we start by separating the two, and we introduce an explicit recursion operator. We then follow the approach described in [Pie02, Section 21.8].

**Example 5.4** A binary tree with natural numbers as leafs can be defined as follows using explicit recursion.

$$\mathit{Tree} = \mu X.\mathbf{struct}\ \mathit{node}(X, X) \mid \mathit{leaf}(\mathbb{N})$$

Now, writing $S = T$ for some basic sort $S$ and an arbitrary sort expression $T$, just introduces $S$ as an alternative name for $T$, whereas recursion is made explicit through the use of fixed points.

We need to take care that equations with mutual recursion are handled appropriately.

**Example 5.5** Consider the following two equations, which are mutually recursive. Note that $h$ is some non-recursive sort.

$$\begin{aligned}\mathbf{sort}\ A &= \mathbf{struct}\ f(B)\\ B &= \mathbf{struct}\ g(A) \mid h\end{aligned}$$

If we just transform those into the following

$$\begin{aligned}\textbf{sort } A &= \mu X.\textbf{struct } f(B)\\ B &= \mu Y.\textbf{struct } g(A) \mid h\end{aligned}$$

we still have to deal with recursion, as adding the fixed points did not change anything. We can overcome this issue by substituting the right hand sides of mutually recursive types, obtaining the following:

$$\begin{aligned}\textbf{sort } A &= \mu X.\textbf{struct } f(\mu Y.\textbf{struct } g(X) \mid h)\\ B &= \mu Y.\textbf{struct } g(\mu X.\textbf{struct } f(Y)) \mid h\end{aligned}$$

Observe that we have unfolded each recursion once in this example, and that we just include a single fixed point. There are cases in which multiple fixed points are needed. Furthermore, as there is a finite number of aliases, the number of substitutions is finite.

Note that dealing explicitly with recursion, as we do in this section, we are able to write expression that are not supported in mCRL2, like the following.

$$\lambda y : \mu X.\textbf{struct } \mathit{pair}(\mathit{first} : \mathbb{N}, \mathit{second} : \mathbb{N}).\mathit{first}(y) + \mathit{second}(y).$$

These kinds of expressions can always be emulated in mCRL2 by introducing an additional alias definition.

### 5.2.1 Types

We have extended the syntax of types with a least fixed-point operator to make the recursion in types explicit. The extended syntax of types is as follows.

**Definition 5.6 (Sort expressions)** The syntax of sort expressions is given as follows:

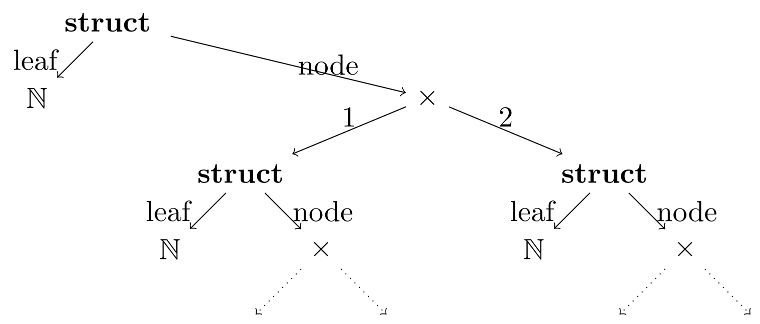
$$\begin{array}{lll} S & ::= & S_{\mathit{Basic}} \mid S \times \cdots \times S \to S \mid \textbf{struct } \mathit{scs}, \cdots, \mathit{scs} \mid \mathit{List}(S) \mid \mathit{Set}(S) \mid \mathit{Bag}(S) \\ & \mid & \mu X.S \\ \mathit{scs} & ::= & f \mid f(\mathit{spj}, \ldots, \mathit{spj}) \mid f?f \mid f(\mathit{spj}, \ldots, \mathit{spj})?f \\ \mathit{spj} & ::= & S \mid f{:}S \end{array}$$

Observe that the above syntax for sort expressions is identical to the one we have given earlier, except for the recursive type $\mu X.S$.

### 5.2.2 Trees of types

The set of (finite and) infinite tree types $\mathcal{T}$ is the greatest fixed point of the generating function described by the above grammar. The universe of this generating function is the set of all finite and infinite trees labelled with $\mathsf{Unit}$, $S_{\mathit{Basic}}$, $\to$, *List*, *Set*, *Bag*, and **struct**, $\times$. We label the edges in the trees in a similar way as before.

**Example 5.7** The tree corresponding to the type $\mu X.\textbf{struct } \ \mathit{leaf}(\mathbb{N}) \mid \mathit{node}(X, X)$ is the following:



The tree extends ad infinitum because of the recursion.

### 5.2.3 Infinite subtyping

The inference system given in the previous section, can be extended to trees of infinite types by adding folding and unfolding rules of the fixed point operator.

$$\frac{S <: T[X := \mu X.T]}{S <: \mu X.T}\ \text{(S-FoldRight)} \qquad \frac{S[X := \mu X.S] <: T}{\mu X.S <: T}\ \text{(S-FoldLeft)}$$

Algorithmically, this is inconvenient, as it is not easily determined when to apply the folding or unfolding rules. Therefore, we resort to a fixed point algorithm for computing the subtyping relation.

Two tree types $S$ and $T$ are in the subtype relation if $(S, T) \in \nu\mathcal{S}(\mathcal{I})$, where $\mathcal{I}$ is the identity relation, and $\mathcal{S}$ is the monotone function $S \in 2^{\mathcal{T}\times\mathcal{T}} \to 2^{\mathcal{T}\times\mathcal{T}}$, defined by

$$\begin{array}{rcl}
\mathcal{S}(R) & = & \{(\mathbb{N}^+, \mathbb{N}), (\mathbb{N}^+, \mathbb{Z}), (\mathbb{N}^+, \mathbb{R}), (\mathbb{N}, \mathbb{Z}), (\mathbb{N}, \mathbb{R}), (\mathbb{Z}, \mathbb{R})\} \\
& \cup & \{(\mathcal{C}(S), \mathcal{C}(T)) \mid (S, T) \in R, \mathcal{C} \in \{\mathit{List}, \mathit{Set}, \mathit{Bag}\}\} \\
& \cup & \{(S_1 \times \cdots \times S_n \to S, T_1 \times \cdots \times T_n \to T) \mid (T_i, S_i), (S, T) \in R, 1 \leq i \leq n\} \\
& \cup & \left\{ \left( \begin{array}{cc} \textbf{struct} f_1(S_{1,1}, \ldots, S_{1,m_1}) & \textbf{struct} f_1(T_{1,1}, \ldots, T_{1,m_1}) \\ \vdots & \vdots \\ |f_n(S_{n,1}, \ldots, S_{n,m_n}), & |f_n(T_{n,1}, \ldots, T_{n,m_n}) \end{array} \right) \middle| \begin{array}{r} 1 \leq i \leq n \\ 1 \leq j \leq m_i \\ (S_{i,j}, T_{i,j}) \in R \end{array} \right\} \\
& \cup & \{(S, \mu X.T) \mid (S, T[X \mapsto \mu X.T]) \in R\} \\
& \cup & \{(\mu X.S, T) \mid (S[X \mapsto \mu X.T]), T) \in R, \forall T'.T \neq \mu X.T', T \notin S_{Basic}\}
\end{array}$$

Note that the rules for $\mu$ types are not symmetric in order to make $\mathcal{S}$ invertible.

The support function corresponding to $\mathcal{S}$ is the following.

$$
\mathit{supp}_{\mathcal{S}}(S,T) = \begin{cases}
\emptyset & \text{if } (S,T) \in \{(\mathbb{N}^+,\mathbb{N}), (\mathbb{N}^+,\mathbb{Z}), \\
 & (\mathbb{N}^+,\mathbb{R}), (\mathbb{N},\mathbb{Z}), (\mathbb{N},\mathbb{R}), (\mathbb{Z},\mathbb{R})\} \\
\{(S,T)\} & \text{if } S = \mathcal{C}(S_1) \text{ and } T = \mathcal{C}(T_1) \\
 & \text{for } \mathcal{C} \in \{\mathit{List}, \mathit{Set}, \mathit{Bag}\} \\
\{(S',T')\} \cup \{1 \le i \le n \mid (T_i, S_i)\} & \text{if } S = S_1 \times \cdots \times S_n \to S' \\
 & \text{and } T = T_1 \times \cdots \times T_n \to T' \\
\{1 \le i \le n, 1 \le j \le m_i \mid (S_{i,j}, T_{i,j})\} & \text{if } S = \textbf{struct} f_1(S_{1,1}, \ldots, S_{1,m_1}) \\
 & \qquad\qquad \vdots \\
 & \qquad\qquad |f_n(S_{n,1}, \ldots, S_{n,m_n}) \\
 & T = \textbf{struct} f_1(T_{1,1}, \ldots, T_{1,m_1}) \\
 & \qquad\qquad \vdots \\
 & \qquad\qquad |f_n(T_{n,1}, \ldots, T_{n,m_n}) \\
\{(S, T'[X \mapsto \mu X.T'])\} & \text{if } T = \mu X.T' \\
\{(S'[X \mapsto \mu X.S'], T)\} & \text{if } S = \mu X.S', \\
 & \text{and } \forall T'.T \neq \mu X.T',\ T \notin S_{Basic} \\
\bot & \text{otherwise}
\end{cases}
$$

**Lemma 5.8** The generating function $\mathcal{S}$ for the subtyping relation is invertible.

**Theorem 5.9** *$\nu\mathcal{S}$ is reflexive and transitive, i.e., $\mathcal{I} \subseteq \nu\mathcal{S}$ and $\nu\mathcal{S} \circ \nu\mathcal{S} \subseteq \nu\mathcal{S}$.*

Using the definitions from this section, an algorithm for computing the greatest fixpoint can be used to decide subtyping, as described before.

## 6 Coercion

In the previous two sections we have described a methodology for subtyping in mCRL2. In an implementation, it is desired that the resulting expression, after typechecking, is strictly typed, *i.e.* that it can be checked for well-typedness without the use of the subtyping rules. In order to achieve this, coercion functions can be applied. Intuitively, a coercion is a function that transforms an expression that is typeable in the type system with subtyping into an expression that is typeable in the type system *without subtyping.*

We introduce the following notation to formalise the translation. $\mathcal{C} :: S <: T$ means that "$\mathcal{C}$ is a subtyping derivation tree whose conclusion is $S <: T$". Likewise we write $\mathcal{D} :: \Gamma \vdash_\Sigma e{:}S$ to mean "$\mathcal{D}$ is a typing derivation whose conclusion is $\Gamma \vdash_\Sigma e{:}S$".

We first introduce the function that, given a derivation $\mathcal{C}$ for the subtyping statement $S <: T$ generates the coercion $[\![\mathcal{C}]\!]$. Observe that $[\![\mathcal{C}]\!]{:}S \to T$. We assume that the following functions are available in the language:

- $\mathit{P2N}{:}\mathbb{N}^+ \to \mathbb{N}$
- $\mathit{P2I}{:}\mathbb{N}^+ \to \mathbb{Z}$

- $\mathit{P2R}{:}\mathbb{N}^+ \to \mathbb{R}$
- $\mathit{N2I}{:}\mathbb{N} \to \mathbb{Z}$
- $\mathit{N2R}{:}\mathbb{N} \to \mathbb{R}$
- $\mathit{I2R}{:}\mathbb{Z} \to \mathbb{R}$
- $\mathit{map}{:}\mathit{List}(S) \times (S \to T) \to \mathit{List}(T)$ for all sorts $S, T$.

Using the above definitions, we define $[\![\_]\!]$ by cases on the final rule used in $\mathcal{C}$. Note that at $\star$ we apply S-Func.

$$\left[\!\!\left[\frac{}{\mathbb{N}^+ <: \mathbb{N}}\text{ (S-P2N)}\right]\!\!\right] = \mathit{P2N}$$

$$\left[\!\!\left[\frac{}{\mathbb{N}^+ <: \mathbb{Z}}\text{ (S-P2I)}\right]\!\!\right] = \mathit{P2I}$$

$$\left[\!\!\left[\frac{}{\mathbb{N}^+ <: \mathbb{R}}\text{ (S-P2R)}\right]\!\!\right] = \mathit{P2R}$$

$$\left[\!\!\left[\frac{}{\mathbb{N} <: \mathbb{Z}}\text{ (S-N2I)}\right]\!\!\right] = \mathit{N2I}$$

$$\left[\!\!\left[\frac{}{\mathbb{N} <: \mathbb{R}}\text{ (S-N2R)}\right]\!\!\right] = \mathit{N2R}$$

$$\left[\!\!\left[\frac{}{\mathbb{Z} <: \mathbb{R}}\text{ (S-I2R)}\right]\!\!\right] = \mathit{I2R}$$

$$\left[\!\!\left[\frac{\mathcal{C} :: S <: T}{\mathit{List}(S) <: \mathit{List}(T)}\text{ (S-List)}\right]\!\!\right] = \lambda x{:}\mathit{List}(S).\mathit{map}([\![\mathcal{C}]\!], x)$$

$$\left[\!\!\left[\frac{\mathcal{C} :: S <: T}{\mathit{Set}(S) <: \mathit{Set}(T)}\text{ (S-Set)}\right]\!\!\right] = \lambda x{:}\mathit{Set}(S). \{y{:}T \mid \exists z{:}S. y = [\![\mathcal{C}]\!](z) \wedge z \in x\}$$

$$\left[\!\!\left[\frac{\mathcal{C} :: S <: T}{\mathit{Bag}(S) <: \mathit{Bag}(T)}\text{ (S-Bag)}\right]\!\!\right] = \text{See Remark 6.1}$$

$$\left[\!\!\left[\frac{\begin{array}{c}\mathcal{C}_1 :: T_1 <: S_1 \\ \vdots \\ \mathcal{C}_n :: T_n <: S_n \\ \mathcal{C} :: S <: T\end{array}}{S_1 \times \cdots \times S_n \to S <: T_1 \times \cdots \times T_n \to T}\ (\star)\right]\!\!\right] = \begin{array}{l}\lambda f{:}S_1 \times \cdots \times S_n \to S.\lambda \vec{x}{:}\vec{T}. \\ [\![\mathcal{C}]\!](f([\![\mathcal{C}_1]\!](x_1), \ldots, [\![\mathcal{C}_n]\!](x_n)))\end{array}$$

$$\left[\!\!\left[\frac{\mathcal{C}_{i,j} :: S_{i,j} <: T_{i,j} \text{ for all } i, j}{\mathit{Struct1} <: \mathit{Struct2}}\text{ (S-Struct)}\right]\!\!\right] = \lambda x{:}\mathit{Struct1}.\mathit{iftree}(x)$$

where $\mathit{Struct1}$ is defined as

$$\begin{array}{l}\textbf{struct} f_1(S_{1,1}, \ldots, S_{1,m_1}) \\ \quad\vdots \\ \quad | f_n(S_{n,1}, \ldots, S_{n,m_n})\end{array}$$

*Struct2* is defined as

$$\begin{aligned}&\textbf{struct} f_1(T_{1,1}, \ldots, T_{1,m_1})\\ &\quad\vdots\\ &|f_n(T_{n,1}, \ldots, T_{n,m_n})\end{aligned}$$

and *iftree*(x) is defined as

$$\begin{aligned}&\mathit{if}(\mathit{is}_{f_1}(x), f_1([\![\mathcal{C}_{1,1}]\!](f_{1,1}(x)), \ldots, [\![\mathcal{C}_{1,m_1}]\!](f_{1,m_1}(x))),\\ &\quad \mathit{if}(\mathit{is}_{f_2}(x), f_2([\![\mathcal{C}_{2,1}]\!](f_{2,1}(x)), \ldots, [\![\mathcal{C}_{2,m_2}]\!](f_{2,m_2}(x))),\\ &\qquad \mathit{if}(\ldots, \ldots, f_n([\![\mathcal{C}_{n,1}]\!](f_{n,1}(x)), \ldots, [\![\mathcal{C}_{n,m_n}]\!](f_{n,m_n}(x)))\ldots)))\end{aligned}$$

After type checking, and applying the coercions, no coercions occur in the syntactic representations of terms; all applications of coercions have been replaced by syntactic elements in mCRL2, with $N2I$, *etc.* as basic elements, extended with lambda abstractions.

**Remark 6.1** Note that the coercion function for bags is not generally defined. Given a subtype relation $S <: T$, we must define a function $\mathit{Bag}(S) \to \mathit{Bag}(T)$, performing the coercion. As an example, we take a look at the case for $S = \mathbb{N}, T = \mathbb{Z}$. We find the following function:

$$\lambda b{:}\mathit{Bag}(\mathbb{N}).\{x{:}\mathbb{Z} \mid \mathit{if}(x < 0, 0, \mathit{count}(I2N(x), b))\}$$

In this function, we use our knowledge about the data type, and the presence of a reverse function of signature $\mathbb{Z} \to \mathbb{N}$. In general, such a function is not provided, hence this solution is not ideal, and only works for the restricted case of the predefined types.

The problem of finding a general coercion function for bags, operating on arbitrary types, is still open. The problem can be formalised as follows. Let $S, T$ be types, such that $S <: T$, and assume a coercion function $C{:}S \to T$, that converts an element of type $S$ to an element of type $T$, such that $C$ is injective. Find a function $f{:}\mathit{Bag}(S) \to \mathit{Bag}(T)$, such that every element of type $\mathit{Bag}(S)$ is converted to an element of type $\mathit{Bag}(T)$.

We also define a translation for type derivations. If $\mathcal{D}$ is a type derivation for $\Gamma \vdash_\Sigma e{:}S$, then translation $[\![\mathcal{D}]\!]$ is a strictly typed term of type $S$.

$$\left[\!\!\left[\frac{x : S \in \Gamma}{\Gamma \vdash_\Sigma x : S}\,(\text{T-Var})\right]\!\!\right] = x^s$$

$$\left[\!\!\left[\frac{f : S_1 \times \cdots \times S_n \to S \in \Sigma}{\Gamma \vdash_\Sigma f : S_1 \times \cdots \times S_n \to S}\,(\text{T-Func})\right]\!\!\right] = f^{S_1 \times \cdots \times S_n \to S}$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Gamma, \vec{x}{:}\vec{S} \vdash_\Sigma e : T}{\Gamma \vdash_\Sigma (\lambda \vec{x}{:}\vec{S}.e) : S_1 \times \cdots \times S_n \to T}\,(\text{T-Abs})\right]\!\!\right] = \lambda \vec{x}{:}\vec{S}.\,[\![\mathcal{D}]\!]$$

$$\left[\!\!\left[\frac{\begin{array}{c}\mathcal{D} :: \Gamma \vdash_\Sigma e{:}S_1 \times \cdots \times S_n \to S\\ \mathcal{D}_1 :: \Gamma \vdash_\Sigma e_1{:}S_1\\ \vdots\\ \mathcal{D}_n :: \Gamma \vdash_\Sigma e_n{:}S_n\end{array}}{\Gamma \vdash_\Sigma e(e_1, \ldots, e_n) : S}\,(\text{T-Appl})\right]\!\!\right] = [\![\mathcal{D}]\!]\,([\![\mathcal{D}_1]\!], \ldots, [\![\mathcal{D}_n]\!])$$

$$\left[\!\!\left[\frac{\begin{array}{c}\mathcal{D}_1 :: \Gamma \vdash_\Sigma e_1 : S_1\\ \vdots\\ \mathcal{D}_n :: \Gamma \vdash_\Sigma e_n : S_n\\ \Gamma, \vec{x}{:}\vec{S} \vdash_\Sigma e : T\end{array}}{\mathcal{D} :: \Gamma \vdash_\Sigma (e \ \mathbf{whr}\ \vec{x} = \vec{e}\ \mathbf{end}) : T}\,(\text{T-Where})\right]\!\!\right] = [\![\mathcal{D}]\!]\ \ \mathbf{whr}\ \vec{x} = [\![\vec{\mathcal{D}}]\!]\ \mathbf{end}$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Gamma, \vec{x}{:}\vec{S} \vdash_\Sigma e : \mathbb{B}}{\Gamma \vdash_\Sigma (\forall \vec{x}{:}\vec{S}.e) : \mathbb{B}}\,(\text{T-Forall})\right]\!\!\right] = \forall \vec{x}{:}\vec{S}.\,[\![\mathcal{D}]\!]$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Gamma, \vec{x}{:}\vec{S} \vdash_\Sigma e : \mathbb{B}}{\Gamma \vdash_\Sigma (\exists \vec{x}{:}\vec{S}.e) : \mathbb{B}}\,(\text{T-Exists})\right]\!\!\right] = \exists \vec{x}{:}\vec{S}.\,[\![\mathcal{D}]\!]$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Gamma, x : S \vdash_\Sigma e : \mathbb{B}}{\Gamma \vdash_\Sigma \{x : S \mid e\} : \mathit{Set}(S)}\,(\text{T-Set})\right]\!\!\right] = \{x : S \mid [\![\mathcal{D}]\!]\}$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Gamma, x : S \vdash_\Sigma e : \mathbb{N}}{\Gamma \vdash_\Sigma \{x : S \mid e\} : \mathit{Bag}(S)}\,(\text{T-Bag})\right]\!\!\right] = \{x : S \mid [\![\mathcal{D}]\!]\}$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Gamma \vdash_\Sigma e : T \quad \mathcal{C} :: \Gamma \vdash_\Sigma T <: S}{\Gamma \vdash_\Sigma e : S}\,(\text{T-Sub})\right]\!\!\right] = [\![\mathcal{C}]\!]\,([\![\mathcal{D}]\!])$$

## 7 Properties of the type system

In this section we investigate some properties of our type system. Note that a lot of properties that one might like to hold, do not actually hold for our type system, because of the liberal data language. Most notably our type system does not ensure a minimal type, *i.e.*, the following property does not hold.

$$\Gamma \vdash_\Sigma e{:}S \implies (\exists T : \Gamma \vdash_\Sigma e{:}T \wedge (\forall U : \Gamma \vdash_\Sigma U \implies T <: U))$$

This property does not hold because of the overloading that is allowed in the language. As an example where this property does not hold, consider the following. Suppose $f{:}S, f{:}T$ are declared as functions. We can than derive that $\Gamma \vdash_\Sigma f{:}S$ and $\Gamma \vdash_\Sigma f{:}T$, but in general $S \not<: T$ and $T \not<: S$. In an implementation it is desired that a type error is given if no *unique* <:*-minimal type* exists.

In general, expositions about type checking also discuss an evaluation relation. An evaluation relation is based on an operational semantics, reducing every closed expression in the language to a value. In the case of mCRL2, this is generally not possible, as the rules for reducing expressions are provided by the user. We therefore omit this property here.

For an algorithm implementing the type system as described in this paper, we want the following properties to hold. Assume that $\mathit{Sub}_\Sigma(S,T)$ is an algorithm deciding whether $S$ is a subtype of $T$, and that $\mathit{Type}_\Sigma(\Gamma,e)$ returns the *unique minimal type* of $e$, if such a type exists, and $\perp$ otherwise.

- $\vdash_\Sigma S <: T \implies \mathit{Sub}_\Sigma(S,T) = \mathsf{true}$, *i.e.*, if we can derive that $S$ is a subtype of $T$ in our inference system, then the algorithm also determines this.
- $\mathit{Sub}_\Sigma(S,T) = \mathsf{true} \implies \vdash_\Sigma S <: T$, *i.e.*, if the algorithm determines that $S$ is a subtype of $T$, then the relation can be inferred.
- $\mathit{Sub}_\Sigma(S,T)$ is defined for all types $S,T$.
- $\mathit{Type}_\Sigma(\Gamma,e) = S \implies \Gamma \vdash_\Sigma e{:}S$, *i.e.*, if the algorithm computes a type $S$, then it must be derivable in our type system.
- $\mathit{Type}_\Sigma(\Gamma,e)$ is defined for all expressions $e$.
- $(\Gamma \vdash_\Sigma e{:}S \wedge \Gamma \vdash_\Sigma e{:}T \wedge S \not<: T \wedge T \not<: S) \implies \mathit{Type}_\Sigma(\Gamma,e) = \perp$, *i.e.*, if an expression $e$ can have multiple, incomparable types, then an error is given by the algorithm.
- $(\Gamma \vdash_\Sigma e{:}S \wedge (\forall \Gamma \vdash_\Sigma e{:}T \implies S <: T) \implies \mathit{Type}_\Sigma(\Gamma,e) = S$, *i.e.*, if an expression $e$ can have multiple, comparable types, then the minimal type is returned.

## 8 Conclusions

We have defined a type inference system for the data language of mCRL2. Our definitions follow the standard approach as described by Pierce [Pie02], but extends the definitions of inference rules because the data language of mCRL2 is richer than the language described in *ibid.* We also provided a coercion function that can be used to move toward a setting with strictly typed expressions. Note that the coercion function is currently not defined for bags of arbitrary types, but is restricted to bags of predefined types. The problem of finding a coercion function for bags of general types is still open.

To obtain a type checker for mCRL2, the approach given in this paper must be extended with definitions for processes, as well as the first order modal $\mu$-calculus. This is mainly an extension of the rules with extra mechanisms for variable binding, as well as imposing restrictions on types, *e.g.*, requiring that guards are Boolean.

In addition, in an implementation one would also like to define restrictions that must be imposed on the data specification. An example of a restriction is that the left hand side of an equation must have the same type as the right hand side of an equation. Checking these kinds of requirements is also a straightforward extension of the approach proposed in this paper.

## References

[BBR09] J.C.M. Baeten, T Basten, and M.A Reniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, November 2009.

[FGK$^+$96] J.C Fernandez, H Garavel, A Kerbrat, R Mateescu, L Mounier, and M Sighireanu. CADP: A protocol validation and verification toolbox. In *Proc. of the 8th conf. on CAV*, pages 437–440, August 1996.

[GMR$^+$09] J F Groote, A H J Mathijssen, M A Reniers, Y S Usenko, and M J van Weerdenburg. Analysis of distributed systems with {mCRL2}. In M Alexander and W Gardner, editors, *Process Algebra for Parallel and Distributed Processing*, pages 99–128. Chapman & Hall, 2009.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of CRL. In A. Ponse, Verhoef C., and S.F.M. van Vlijmen, editors, *ACP'94*, pages 26–62. Springer, 1995.

[GR10] J.F. Groote and M.A. Reniers. *Modelling and Analysis of Communicating Systems.* unpublished, 2010.

[GW05] J.F. Groote and T.A.C. Willemse. Model-checking processes with data. *Science of Computer Programming*, 56(3):251–273, May 2005.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[Koz83] D Kozen. Results on the propositional $\mu$-calculus. *Theor. Comp. Sc.*, 27:333–354, 1983.

[mCR] mCRL2 website (`http://www.mcrl2.org`).

[NK04] R Nederpelt and F Kamareddine. *Logical Reasoning: A First Course.* King's College Publications, 2004.

[Pie02] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[Tar55] A. Tarjan. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[Wee07] M. van Weerdenburg. An account of implementing applicative term rewriting. In *WRS 2006*, volume 174 of *ENTCS*, pages 139–155, 2007.