MSR 2.0: Language Definition and Programming Environment

Iliano Cervesato*

November 2011
CMU-CS-11-141
CMU-CS-QTR-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Carnegie Mellon University, Qatar campus.
The author can be reached at iliano@cmu.edu.

Abstract

This report defines the syntax and semantics of the specification language *MSR 2.0*, and gives requirements for a run-time environment for it. Specifically, it defines the concrete syntax of the language and formalizes its typing and execution semantics at an abstract level. It also describes programming environment functionalities such as type reconstruction and facilities for controlling execution. *MSR 2.0* is a specification language based on first-order multiset rewriting modulo equations, dependent types, and subsorting. This report is meant to act as its "official" definition, as various subsets have appeared in previous publications, for example [6, 7]. It has been used extensively for studying cryptographic protocols [5, 8, 9, 10, 12, 13, 15] and especially Kerberos 5 [1, 2, 3, 4, 16, 17]. It was also used experimentally for modeling bio-molecular systems [11]. *MSR 1.0*, the precursor of this version, was also used in foundational studies for crypto-protocols [14, 19]. An implementation of *MSR 2.0* which adheres to the definition presented in this report has been written in Maude [18, 21].

Partially supported by QNRF under grant NPRP 09-667-1-100. This work was started while the author was working at the Naval Research Laboratory under contract N00173-00-C-2086. Parts were completed during an extended visit at Princeton University.

Keywords: MSR 2.0, multiset rewriting, specification language, language definition, syntax, typing, execution, type reconstruction.

Contents

Chapter 1

Lexicon

Throughout this report, we refer to *MSR 2.0* as *MSR*. This chapter describes the lexicon of *MSR* using regular expressions. Meta-symbols (*e.g.,* '[', '(' and '+') will be printed as normal mathematical text. ASCII characters or their traditional rendering (for non-printable characters) will be written using a `typewriter` font or in hexadecimal preceded by the string '0x'.

1.1 Spaces

Spaces (non-terminal *sp*) are strings of one or more white space (␣, ASCII 0x20), tabulation (`\t`, ASCII 0x09), new line (`\n`, OS-dependent combination of `\r` and `\f`), carriage return (`\r`, ASCII 0x13), and form feed (`\f`, ASCII 0x12).

sp = [␣`\t\n\r\f`]+

1.2 Comments

Comments are of two types:

Single-line comments start with the symbol `%` and extend to the end of the line.

> System directives piggyback on this syntax and have the form `%`*token args*, where *token* is a reserved symbol that has meaning for the interpreter only and *args* are the arguments it may take. Directives instruct the run-time system to interpret certain symbols as infix with a certain precedence, for example.

Multi-line comments start with the symbol `%{` and extend to the next matching occurrence of `}%`.

> Multi-line comments can be nested and therefore should be ended by balanced occurrence of `}%`. Unterminated multi-line comments should be reported as errors (in particular they are not implicitly closed when encountering the end of the current file). Unbalanced occurrences of `}%` should also be reported as errors.

A comment is either a single-line or a multi-line comment.

Comments can occur at any point in a file, and are equivalent to the empty string. For example, '`my%{xxx}%Id`' is equivalent to the identifier '`myId`'. A single-line comment (together with the terminating `\n`) can similarly be excised.

singlelineComment = `%.*`
multilineComment = `%{[.\n]*}%`
comment = *singlelineComment* + *multilineComment*

1.3 Special Characters

Special characters are not allowed as part of identifiers. They include a minimal set of punctuation and grouping symbols.

specialChar = `[%.)(}{]`

1.4 Reserved Symbols

A number of otherwise valid identifiers are reserved operators of *MSR*. They include the following, each with a brief description:

`include`	File inclusion
`module`	Module declaration
`import`	Imported symbols of a module
`export`	Exported symbols of a module
`*`	Import/Export wild-card
`:`	Label separator
`type`	Type classifier
`state`	Multiset classifier
`->`	Functional type
`<:`	Subsort declaration
`_`	Irrelevant variable
`=`	Equation symbol
`for`	Anchored role header
`forall`	Universal quantifier
`exists`	Existential quantifier
`=>`	Rewrite rule separator
`,`	State element separator
`empty`	Empty multiset
`;`	Guard separator
`if`	Guard separator (alternate syntax)
`[`*id*`]`	Reserved for future use

1.5 Identifiers

An identifier is a non-empty string of printable ASCII characters that does not include any special character, is not a space, a comment, or a reserved symbol.

id = $[0x21 - 0x7e]+$

Identifiers are separated by spaces or special characters. So, '`my Id`' contains two identifiers ('`my`' and '`Id`'), and similarly for '`my(Id)`'.

1.6 Errors

An error message should be output whenever one of the following situations occur:

1. the lexical analyzer encounters an occurrence of `)`, `}` or `}%` that does not match a previous occurrence of `(`, `{`, or `%{`, respectively;

2. the lexical analyzer reaches the end of a file without being able to balance all occurrences of `(`, `{`, or `%{` with corresponding occurrences of `)`, `}` or `}%`, respectively.

The error message should point to the first opening/closing symbol without a corresponding closing/opening symbol.

Chapter 2

Syntax

This chapter describes the syntax of *MSR* specifications on the basis of the lexical entities presented in Chapter 1. We use standard BNF notation for this purpose. Meta-symbols will be rendered as normal mathematical text (*e.g.,* '::=' or '|'). The meta-symbol '·' denotes the empty string. Terminals are displayed using a `typewriter` font. Non-terminals are written using a slanted font, such as the already defined *id*. Non-terminals used before they are defined will be written in gray (*e.g.,* *subtp*). Occasionally, portions of syntax can be reconstructed from the context or by default rules, or are otherwise optional. We enclose them in square braces [...] and comment on it in the text.

2.1 Organizing Specifications

An *MSR* specification can be split among a number of files. The `include` directive is provided to embed the contents of a file into another file: upon encountering '`include` *file*' the compiler replaces this statement with the contents of file *file*. `include` statements can be nested, but should not be circular.

Although it is natural to break specifications in files at the module level, we adopt the more liberal policy of allowing `include` directives at any point in a specification. This corresponds to having parsing proper begin *after* all the `include` directives have been expanded.

2.2 Module Infrastructure

An *MSR* specification is organized as a collection of modules, possibly interspersed with items (declarations, equations, roles, or state objects — see below) and commands. Commands are mostly used at the top level of the *MSR* tool and their use in a module is not encouraged.

A module is introduced by the keyword `module`, labeled by a unique identifier, and defined by a header and a body. The header lists symbols imported from other modules and exported outside the defining module. The body of a module declares the symbols used within the module (with the exception of those imported), the equations that govern them, and the rules that operate on them.

A module starts with the keyword `module` and ends either at the end of the specification, or upon encountering the keyword `module` that indicates the beginning of another module. Modules cannot be nested.

MSRspec ::= · | *module* *MSRspec* | *item* *MSRspec* | *command* *MSRspec*	*S*
module ::= `module` *id* *header* *body*	*M*

The header lists symbols imported from other modules and exported outside the defining module. Import

declarations consist of the keyword `import`, the importing module name, and the imported items, which are either a single body item or a comma-separated list of labels. They end with a period. Exported items consist of the keyword `export`, a comma-separated list of labels, and a period.

header ::= *imports exports*	*H*
imports ::= · | *imports* `import` *id* `*.` | *imports* `import` *id* id^+`.` | *imports* `import` *id item*	*I*
exports ::= · | `export *.` | *exports* `export` id^+`.`	*X*
id^+ ::= *id* | *id*`,` id^+	

Import lists will generally mention only labels (*id*). A complete item, inclusive of the label and the type, equation or role is available for when a module is intended for separate compilation. When labels only are given, the corresponding items shall be accessed from the referenced module. Both imports from a same module and exports can be grouped or broken in *import* and *export* declarations at the specifier's leisure. For convenience, the wild-card `*` can be used in place of declarations to specify that a module shall import all symbols exported by another module, or that it exports all symbols it defines.

The scope of an import or export declaration is the whole module.

The body of a module is a sequence of body items (*item*), which consist of declarations (*decl*), equations (*equation*), definitions (*definition*), and roles (*role*), in arbitrary order. As will become apparent in the sequel, items consist of a label, a colon ":", and typing information, an equation or a role. The label and the colon are optional except for typing information. An identifier, *i.e.,* a label, must be declared before its first use in an equation or a role. With the exception of roles, body items end with a period.

item ::= *decl*`.` | *equation*`.` | *definition*`.` | *role*	*b*
body ::= · | *body item*	*B*

The scope of an item extends to the end of the present module. It is expected that each identifier used in a module be declared in the module, and that no two declarations declare the same identifier.

Commands are used to interact with and guide the *MSR* tool. See the user documentation of the tool, currently [21], for the list of available commands and their effects.

command ::= ···	

2.3 Declarations

A declarations is either a type declaration (*tpDecl*), a subtype declaration (*subtpDecl*), or an object declaration (*objDecl*). Each of these forms will be terminated by a period.

decl ::= *tpDecl* | *subtpDecl* | *objDecl*	*δ*

It is expected that symbols have a unique declaration in a module, be it in the *header* through an *import* statement, or in the body as a proper declaration.

2.3.1 Type Declarations

Type declarations declare type constructors. Because *MSR* is dependently typed, we define types before type declarations themselves.

A base type is either the reserved symbol `state` or an identifier possibly applied to a sequence of terms (its arguments or indices). A type is either a base type or a dependent function type, *i.e.,* a type prefixed by an object declaration.

baseTp ::= `state` | *id* | *baseTp term*	*a*
tp ::= *baseTp* | *tp* `->` *tp* | `{`*objDecl*`}`*tp*	τ

A dependent function type will have the form $\{x:\tau_1\}\tau_2$: the object declaration $x:\tau_1$ binds every free occurrence of x in τ_2. As often done, we abbreviate this expression as τ_1 `->` τ_2 whenever x does not occur free in τ_2. The type `state` is used to classify multisets and their elements.[1]

Kinds declare type constructors. A kind is either the symbol `type`, or a dependent kind, *i.e.,* a kind prefixed by an object declaration. As for types, we use the abbreviation τ `->` κ for a kind $\{x:\tau\}\kappa$ where x does not occur in κ.

kind ::= `type` | *tp* `->` *kind* | `{`*objDecl*`}`*kind*	κ

The kind `type` will be used to classify types.

A type declaration is a pair consisting of an identifier and a kind. They are separated using a colon, and terminated by a period (see *item*).

tpDecl ::= *id* `:` *kind*

Any valid identifier can be declared by a type declaration, with the exception of the reserved symbols of *MSR*, including the underscore "`_`".

Type declarations can be accompanied by directives of the form `%name` τ X that instruct the compiler to print variables of type τ it may produce during type reconstruction or execution using the symbol X followed by progressive numbers. If no `%name` directive is given, the default is to use `X` followed by a progressive number.

tpDirective ::= `%name` *id id*

Other directives may be supported by the implementation. See the tool documentation [21] for further information.

2.3.2 Subtype Declarations

A subtype is a pair of types separated by `<:` and possibly prefixed by a number of object declarations enclosed in braces. A subtype declaration is a subtype terminated by a period (see *item*).

subtp ::= *tp* `<:` *tp* | `{`*objDecl*`}`*subtp*	σ
subtpDecl ::= [*id* `:`] *subtp*	

In most cases, the object declarations prefixing a subtype will be reconstructed, either in part or in full. In the former case, the type is left out and shall be inferred. In the latter case, the identifier is omitted as well. By convention, only identifiers beginning with a capital letter or underscore ("`_`") can be assumed to have been implicitly declared in this way.

[1] `state` is designated as a type mostly to simplify the structure of the current Maude implementation of *MSR* [18, 21]. It can also be designated, possibly more accurately, as a kind. While the choice is important from a type-theoretic perspective, it has little effect for a user of the language as described in this document. Future extensions may however require changing this classification.

2.3.3 Object Declarations

Object declarations associate an identifier with a type. The pair is separated by a colon and terminated by a period at the top level or inside modules (see *item*).

objDecl ::= *id* [`:` *tp*]	*o*

Any valid identifier can be declared by a type declaration, with the exception of the reserved symbols of *MSR*, including the underscore "`_`".

Within a rule, the type part of an object declaration can often be reconstructed by the context. It is omitted in those cases. The typing information shall be present in module-level declarations or at the top level.

An object declarations can be accompanied by a directives describing non-standard ways to parse occurrences of this symbol. A unary constant f can be declared either prefixed or postfixed by means of the directives `%prefix` and `%postfix`. They accept a precedence as a second argument. Binary symbols can be requested to be infix of a declared precedence and associativity (`left`, `right` or `none`).

Precedences are numbers between 10,000 and 99,999. Lower and higher precedences are reserved to describe the interpreted operators of *MSR*. Symbols with higher precedences bind more tightly than symbols with lower precedences. As a default, the juxtaposition of symbols is to be considered infix, left-associative, and with precedence 5,000. Precedences can be overridden using parentheses.

objDirective ::= `%prefix` *id prec* | `%postfix` *id prec* | `%infix` *id prec assoc*
prec ::= $[\texttt{1}-\texttt{9}][\texttt{0}-\texttt{9}]^5$
assoc ::= `left` | `right` | `none`

These directives are available for the convenience of the user. All symbols can be rewritten in standard prefix form, possibly with the addition of extra pairs of parentheses where necessary. In the sequel, we will assume that this expansion phase has taken place.

2.4 Equations

An equation relates two terms with the symbol `=`. The pair can optionally be prefixed by a sequence of declarations for the variables occurring in it, although this information can generally be reconstructed. Another optional component is a label that identifies this equation for export. Equations end with a period (see *item*).

equation ::= [*id* `:`] *eq*
eq ::= *term* `=` *term* | [`forall` *objDecl*] *eq* — *E*

MSR does not impose a format on the left-hand side and right-hand side of an equation. However, such formats may be expected from the term rewriting system that handles equations in an implementation. Therefore, it is advisable to use standard formats for equations, for example left linear. Formatting error messages will be relayed by the underlying term rewriting system.

2.5 Definitions

A definition introduces an abbreviation for a term. A definition can be parametric. Formal parameters are specified in a parameter list with each element consisting of an identifier and an optional type (subject

to reconstruction). Parentheses are not needed if the type is omitted. A parameter that does not appear in the body of the definition (but only in omitted typing information) can be omitted whenever it can be reconstructed.

param ::= · | *param* [(*objDecl*)]	$\vec{o}$
definition ::= *id param* `:=` *term*	D

A definition associates an identifier followed by zero or more formal parameters with the term it is intended to abbreviate. Definitions end with a period (see *item*). A definition cannot be recursive. A defined constant is used exactly like a declared constant.

2.6 Roles and Rules

Roles are sequences of rules bound together by a declaration. Roles may declare local symbols that will be called existential variables. A rule consists of a guard, a left-hand side and a right-hand side. These three entities are defined over the notion of multiset, which in turn relies on terms. We will now define these objects from the bottom up.

2.6.1 Terms

A term is either an identifier, or a term applied to another term. Juxtaposition is infix, left-associative, and with precedence 5,000.

term ::= *id* | *term term* | (*term* : *tp*)	t

The last production annotates a term with a certain type. It is included for debugging purposes upon encountering a type reconstruction error. This is not intended as a substitute for providing missing declarations. It is always possible to infer the type of a term given sufficient declarations.

2.6.2 Multisets

A multiset is represented in *MSR* as a comma-separated list of terms. For convenience, the empty multiset can be represented either as the empty string (in which case no surrounding comma is required) or as the terminal `empty`.

mset ::= · | `empty` | *mset* `,` *term*	m

2.6.3 Rules

A rule has a core consisting of a *left-hand side*, a *right-hand side* and optionally a *guard*. This core is prefixed by a number of optional object declarations introduced by the keyword `forall`. The left-hand side and the guard are multisets. The right-hand side is a multiset possibly prefixed by a sequence of object declarations introduced by the keyword `exists`.

rhs ::= *mset* | `exists` *objDecl rhs*	*rhs*
rule ::= *mset* `=>` *rhs* | *mset* `;` *mset* `=>` *rhs* | *mset* `=>` *rhs* `if` *mset* | [`forall` *objDecl*] *rule*	r

The base syntax for a rule is "*g* `;` *lhs* `=>` *rhs*". An alternative way to write the same expression is "*lhs* `=>` *rhs* `if` *g*". Whenever the guard *g* is empty, it can be dropped, simply writing "*lhs* `=>` *rhs*".

The scope of an `exist` declaration extends to the end of the right-hand side, and that of a `forall` declaration, either explicit or implicit, extends to the end of the current rule.

In many cases, the `forall` declarations in a rule can be reconstructed, either in full or in part. In the former case, the type is left out and shall be inferred. In the latter case, the identifier is omitted as well. By convention, only identifiers beginning with a capital letter or underscore ("_") can be assumed to have been implicitly declared in this way.

`exists` declaration can be reconstructed only in part, *i.e.,* only the type portion of these declarations can be omitted.

2.6.4 Roles

A rule sequence, *rule**, is a period-separated list of rules, possibly interspersed with object declarations introduced by the `exists` keyword. These will serve to bind identifiers local to the role. An optional label may prefix a rule.

A role is given by a rule sequence enclosed in curly braces, and prefixed by a label and either an object declaration or by the form "`for` *id*". These prefixes are intended to designate the owner of the role.

rule* ::= $\cdot$ | `empty` | [*id* `:`] *rule* `.` *rule** | `exists` *objDecl rule**	*rs*
role ::= *id* `:` `forall` *objDecl* `{` *rule** `}` | *id* `:` `for` *id* `{` *rule** `}`	ρ

The scope of an `exists` declaration extends to the end of the rule set. The scope of the `forall` declaration of a role, when present, extends to the entire rule set that it qualifies.

2.7 Parsing

The parser shall recognize valid *MSR* specifications according to the above grammar and build appropriate data structures to support subsequent functionalities.

Whenever it is established that the input files do not satisfy this grammar, an error message should be output. It shall be accompanied by information, as precise as possible, about the location of the error.

Chapter 3

Type Reconstruction

Each constant occurring in an *MSR* module is to be interpreted as having been introduced by a declaration, either in the header of the current module, or in the earlier part of this module, or in a dependent prefix, or in an equation as a `forall` declaration, or in a definition as a formal parameter, or in a role (as the leading `forall` or through `exists`), or inside the rule itself as a `forall` or `exists` declaration. The latter are called the local variables of the rule.

As a matter of readability, convenience to the user and usability of the tool, a number of declarations and typing information can be omitted. An *MSR* implementation is expected to either reconstruct this omitted data, when this can be done unambiguously according to the rules below, or issue detailed error messages so that the user can add sufficient text to help the compiler reconstruct omitted parts.

The acceptable omissions fall into the following two classes:

- Implicit bindings comprise
  - the `forall` declarations at the head of a rule or equation,
  - the dependent prefixes of a type, subtype, or kind, and
  - the formal parameters of a definition as long as they do not appear in the body.
- Type part of an object declaration.

The omitted dependent prefixes of a type or kind declaration and the omitted formal parameters of a definition also lead to the omission of the corresponding parameters when these symbols are used.

A detailed account of type reconstruction in the presence of dependent types can be found in [20].

3.1 Reconstruction of `forall` Bindings

The binders `forall` $X : \tau$ occurring in a rule or equation can be omitted as long as

- the implicitly bound variable starts with a capital letter `[A-Z]` or the underscore "_", or is the underscore symbol by itself, which has a special interpretation (see below), and
- type τ can be reconstructed.

Upon encountering an undeclared symbol X starting with a capital letter or the underscore, the compiler shall extend the rule or equation it occurs in with the prefix `forall` X (the still implicit type is reconstructed at a later stage, or a type variable α, to be instantiated subsequently, can be inserted).

Each occurrence of the underscore symbol "_" stands for a different implicitly declared variable. Therefore, the compiler shall replace each occurrence with a different new variable X, and extend the current rule with the prefix `forall` X, as in the previous case. Because of this special interpretation, the underscore symbol shall not be declared.

The reconstruction algorithms outlined in this section apply only within a rule or an equation, and not elsewhere. It shall also be noted that, in general, this simple syntactic binder reconstruction needs to be alternated with the more complex reconstruction of omitted types, as an omitted type may make use of a yet undeclared variable.

3.2 Reconstruction of Dependent Prefixes

Every symbols occurring in a base type shall be declared before its first use. However, since many of these declarations are bookkeeping dependent prefixes of the type, subtype, or kind they appear in, it is convenient to omit them whenever they can be reconstructed.

We adopt the same syntax as in the case of omitted `forall` declarations: implicitly declared variables occurring in a type shall begin with a capital letter `[A-Z]` or the underscore "_", or be the underscore. The missing binder for symbol X is added at the front of the type, subtype or kind as an object declaration $\{X\}$ with its type omitted (or temporarily expressed as a type variable). More precisely, we extend

- a type τ into $\{X\}\tau$,
- a subtype σ into $\{X\}\sigma$, and
- a kind κ into $\{X\}\kappa$.

Each occurrence of the underscore "_" is first replaced with a different identifier before this extension step.

Whenever a symbol starting with a capital letter or the underscore occurs in a type within a rule, it may need to be expanded as either a `forall` declaration or a dependent prefix. In general, the latter is preferred if it occurs within a single type after all the reconstruction steps have been performed. A `forall` declaration is necessary only when this symbol occurs in two or more different types.

Other considerations are as in the case of implicit `forall` declarations.

3.3 Reconstruction of Omitted Types

Object declarations "$x : \tau$" can be abbreviated as "x" whenever the type τ can be reconstructed from the context. Type reconstruction information is obtained from previous declarations. For example, if a specification contains an explicit or reconstructed declaration $f : \tau \to \tau'$ for a symbol f, and a type $\{X\}(\ldots fX \ldots)$ where the type of X has been omitted, then it can be deduced that the type of X shall be τ.

Type reconstruction is usually done by inserting type variables in place of the omitted types, and collecting and propagating constraints until a single solution emerges (*success*), the constraints are found to be inconsistent (*fatal error*), or all possible constraint simplifications have been attempted but constraints remain (*underspecification error*). In the latter case, the author of the specification shall be invited to add typing information and run the checker again.

3.4 Subtypes

Whenever a type subject to reconstruction belongs to a subtype hierarchy, the result shall be the lower bound of all types occurring in the constraints, subject to the usual variance and contravariance conditions.

For example, given a subtype declaration τ `<:` τ' and a variable whose type can be reconstructed as either τ or τ', τ shall be preferred. If this lower bound is not unique, then an underspecification error shall be returned.

3.5 Definitions

Every symbol occurring in the body of a definition shall either appear among its formal parameters or be declared earlier in the module. These symbols cannot be reconstructed as formal parameters since their occurrence order would be up to the implementation: a user would not know how to specify actual values when using the definition. However, the types of the formal parameters are subject to reconstruction. Moreover, whenever the reconstructed types mention variables that do not occur in the body, hence new formal parameters, they are subject to full reconstruction since the implementation can decide on an order both at the definition and use level, without user input.

3.6 Implicit Arguments

Whenever an identifier declaration relies on omitted typing information, any use of this identifier shall omit all arguments corresponding to this omitted typing information. For example, if f is declared of type $a\ X$ `->` b (by means of the declaration f `:` $a\ X$ `->` b) where the type of X is kept implicit, then every use of f shall take the form $f\ t$ for some term t of type $a\ t'$ for an appropriate term t'. If instead f is declared of type `{`X`:`c`}`$a\ X$ `->` b (by means of the declaration f `:` `{`X`:`c`}`$a\ X$ `->` b) with the type of X given explicitly, then every use of f can only be of the form $f\ t'\ t$ where t' has type c and t has type $a\ t'$.

During reconstruction, these omitted arguments shall be reconstructed as well.

3.7 Output of Reconstructed Information

It is useful to have the implementation return successful output information in at least two modes:

Verbose. In this mode, the compiler shall display outputs that include all the reconstructed binding and typing information. This mode is useful mostly for debugging purposes.

Normal. Under normal usage, the compiler shall display outputs without any reconstructed information. For example, if the specification declares a function $f : \tau' \to \tau''$ that the type reconstruction phase completes as $f : \{X : \tau\}\tau' \to \tau''$, uses of f shall have the form $f\ t$ (for t of type τ'), not $f\ x\ t$ for an appropriate x of type τ.

Intermediate modes may be useful if it is found that the verbose mode displays overwhelming amounts of information, but the normal mode is too succinct to permit effective debugging.

3.8 Errors

The implementation shall report an error whenever omitted parts of an *MSR* specification cannot be reconstructed unambiguously. It is preferable to output modules and other declarations as soon as they have been fully reconstructed in order to narrow down the portion of a program that requires user attention. Error messages should be as informative as possible.

Chapter 4

Type Checking

This chapter identifies those *MSR* specifications that shall be viewed as sensible by requiring symbols to be used consistently with their declaration. This is achieved by laying out a typing semantics on top of the *MSR* syntax defined in Chapter 2. We assume that implicit information has been reconstructed in full as described in Chapter 3. Those specifications and components that satisfy the typing semantics will be called valid.

We present the typing semantics of *MSR* using judgments defined by typing rules, following the format of Structured Operational Semantics. Whenever a meta-variable of any postulated syntactic category appears below, it shall be implicitly assumed that it stands for a well-formed expression within that syntactic category.

4.1 Substitutions

The meta-level operation of substitution is used in many of the rules needed to validate judgments involving dependent types. It is also used during execution, as discussed in Chapter 5.

The capture-avoiding substitution of a term t for a variable x in an object o of the appropriate syntactic category is uniformly denoted $[t/x]o$ for each class of objects. In case of ambiguity, the specific category of o will be indicated using a superscript, as in $[t/x]^{mset}rhs$. Substitution is defined for all syntactic categories except modules and their headers, bodies, definitions and roles.

Terms

Term substitution, $[t/x]t'$ propagates across the structure of the term t' until an identifier is found. If this identifier is the variable x, it is replaced with t, otherwise it remains unchanged.

$$\begin{array}{lcl} [t/x]y & = & \begin{cases} t & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\ [t/x](t'\ t'') & = & ([t/x]t')\ ([t/x]t'') \\ [t/x](t' : \tau) & = & (([t/x]t') : ([t/x]\tau)) \end{array}$$

Kinds, Types, and Subtypes

Substitution propagates across the structure of kinds, types and subtypes until a term is encountered, where substitution on terms is invoked. The substitution meta-operation is capture-avoiding: whenever it traverses a binding construct, such as a dependent type, it implicitly renames the bound variable to a symbol different

from x if needed.

Kinds:	$[t/x]\texttt{type}$	$=$	$\texttt{type}$
	$[t/x](\{y:\tau\}\kappa)$	$=$	$\{y:[t/x]\tau\}([t/x]\kappa)$
Types:	$[t/x]\texttt{state}$	$=$	$\texttt{state}$
	$[t/x]y$	$=$	y
	$[t/x](a\ t')$	$=$	$([t/x]a)\ ([t/x]t')$
	$[t/x](\{y:\tau\}\tau')$	$=$	$\{y:[t/x]\tau\}([t/x]\tau')$
Subtypes:	$[t/x](\tau <: \tau')$	$=$	$([t/x]\tau) <: ([t/x]\tau')$
	$[t/x](\{y:\tau\}\sigma)$	$=$	$\{y:[t/x]\tau\}([t/x]\sigma)$

Equations

Substitution over equations traverses the universal quantifiers, renaming their bound variables as needed, and then invokes term substitution on each side of the equality symbol.

$[t/x](\texttt{forall}\ y:\tau.E)$	$=$	$\texttt{forall}\ y:([t/x]\tau).([t/x]E)$
$[t/x](t'=t'')$	$=$	$([t/x]t')=([t/x]t'')$

Multiset, Rules, and Rule Sequences

Substitution over multisets reduces to the term substitution over each member term. Substitution over right-hand sides traverses the existential quantifiers, renaming their bound variables as needed, and then invokes multiset substitution. Substitution over rules traverses the universal quantifiers, renaming their bound variables as needed, and then invokes multiset substitution on the guard and left-hand side, and right-hand side substitution on the right-hand side of each rule. Substitution over a rule sequence reduces to rule substitution over each embedded rule. The existential quantifiers are traversed, renaming their bound variables as needed.

Multisets:	$[t/x]\texttt{empty}$	$=$	$\texttt{empty}$
	$[t/x](m,t')$	$=$	$([t/x]m),([t/x]t')$
Right-hand sides:	$[t/x]m$	$=$	$[t/x]^{mset}m$
	$[t/x](\texttt{exists}\ y:\tau.rhs)$	$=$	$\texttt{exists}\ y:([t/x]\tau).([t/x]rhs)$
Rules:			
	$[t/x](m;m' => rhs)$	$=$	$([t/x]m);([t/x]m') => ([t/x]rhs)$
	$[t/x](\texttt{forall}\ y:\tau.r)$	$=$	$\texttt{forall}\ y:([t/x]\tau).([t/x]r)$
Rule sequences:	$[t/x]\texttt{empty}$	$=$	$\texttt{empty}$
	$[t/x](r.rs)$	$=$	$([t/x]r).([t/x]rs)$
	$[t/x](\texttt{exists}\ y:\tau.rs)$	$=$	$\texttt{exists}\ y:([t/x]\tau).([t/x]rs)$

4.2 Typing Judgments

The typing semantics of *MSR* is specified on the basis of the following judgments, which follow closely the syntax described in Chapter 2. The definition of these judgments is given in the next section.

$\vdash S$ spec	*S is a valid specification*
$S \vdash M$ module	*M is a valid module in specification S*
$S \vdash I$ imports	*I is a valid import list w.r.t. specification S*
$B \vdash X$ exports	*X is a valid export list w.r.t. extended body B*
$\vdash \Gamma$ context	*Γ is a valid context*
$\Gamma \vdash B$ body	*B is a valid body in context Γ*
$\Gamma \vdash \kappa$ kind	*κ is a valid kind in context Γ*
$\Gamma \vdash \tau : \kappa$	*τ is a valid type of kind κ in context Γ*
$\Gamma \vdash \sigma$ inst	*σ is a valid subtype in context Γ*
$\Gamma \vdash E$ eq	*E is a valid equation in context Γ*
$\Gamma \vdash t : \tau$	*t is a valid term of type τ in context Γ*
$\Gamma \vdash m$ mset	*m is a valid multiset in context Γ*
$\Gamma \vdash rhs$ rhs	*rhs is a valid right-hand side in context Γ*
$\Gamma \vdash r$ rule	*r is a valid rule in context Γ*
$\Gamma \vdash rs$ rules	*rs is a valid rule sequence in context Γ*
$\Gamma \vdash \rho$ role	*ρ is a valid role in context Γ*

4.3 Typing Rules

4.3.1 Specifications

A valid *MSR* specification ($\vdash S$ spec) is a possibly empty sequence of modules. Each module shall be valid in the context provided by the subspecification that precedes it. This is in order to verify the validity of import statements.

$\vdash S$ spec	*S is a valid specification*

$$\frac{}{\vdash \cdot \text{ spec}}\ \textbf{vS\_·} \qquad \frac{\vdash S \text{ spec} \quad S \vdash M \text{ module}}{\vdash S\ M \text{ spec}}\ \textbf{vS\_M}$$

This judgment depends on the judgment validating a module w.r.t. a specification ($S \vdash M$ module), described below.

4.3.2 Modules

A module M consisting of an identifier id, an import list I, an export list X and a body B is valid w.r.t. a specification S when the import list is valid w.r.t. S, the body prefixed with the inlining of I is valid, and this same entity includes the export list X.

$S \vdash M$ module	*M is a valid module in specification S*

$$\frac{S \vdash I \text{ imports} \qquad \cdot \vdash \text{inline}_S(\overline{I}), B \text{ body} \qquad \text{inline}_S(\overline{I}), B \vdash \overline{X} \text{ exports}}{S \vdash \texttt{module}\ id\ I\ X\ B \text{ module}}\ \textbf{vM}$$

This judgment depends on the judgment validating an import list w.r.t. a specification ($S \vdash I$ imports), the judgment validating the body of a module, extended with all the items it recursively imports ($\Gamma \vdash B$ body), the function $\text{inline}_S(I)$ that retrieves the items corresponding to an import list from the modules where they

were originally introduced, the auxiliary functions $\overline{I}$ and $\overline{X}$, and the judgment that validates an export list w.r.t. the extended body ($B \vdash X$ exports).

The rest of this section defines these judgments, with the exception of the validation of the extended body, which is specified in the next section. For convenience, we first define the auxiliary functions $\overline{I}$ and $\overline{X}$, that break an import and an export list so that each `import` or `export` statement contains one label (or the wild-card $*$, or an actual body item cum label $c : b$).

$$\begin{cases} \overline{\cdot} & = & \cdot \\ \overline{I, \texttt{import}\ id\ *} & = & \overline{I}, \texttt{import}\ id\ * \\ \overline{I, \texttt{import}\ id\ b} & = & \overline{I}, \texttt{import}\ id\ b \\ \overline{I, \texttt{import}\ id\ (cs, c)} & = & \overline{I, \texttt{import}\ id\ cs}, \texttt{import}\ id\ c \\ \overline{I, \texttt{import}\ id\ c} & = & \overline{I}, \texttt{import}\ id\ c \end{cases} \qquad \begin{cases} \overline{\cdot} & = & \cdot \\ \overline{X, \texttt{export}\ *} & = & \overline{X}, \texttt{export}\ * \\ \overline{X, \texttt{export}\ (cs, c)} & = & \overline{X, \texttt{export}\ cs}, \texttt{export}\ c \\ \overline{X, \texttt{export}\ c} & = & \overline{X}, \texttt{export}\ c \end{cases}$$

The function $\text{inline}_S(I)$ recurses over the list I. It is assumed that I has been expanded using the function $\overline{I}$. For reasons of succinctness, we will treat the cases where an element refers to a label c and a body item $(c : b)$ together, writing $c[: b]$, which shall be understood as the former if "$: b$" is absent, and as the latter otherwise. For each element in I, $\text{inline}_S(I)$ produces the corresponding item from the importing module. This item can be found in either its body, or in its import list, in which case $\text{inline}_S(\_)$ calls itself recursively. When encountering an `import` $id\ *$ declaration, it imports the body of module id and calls itself recursively on its import list.

$$\begin{cases} \text{inline}_S(\cdot) & = & \cdot \\ \text{inline}_S((\texttt{import}\ id\ *), I) & = & \text{inline}_{S'}(I'), B, \text{inline}_S(I) \\ \quad \text{where } S = S', (\texttt{module}\ id\ I'\ X\ B), S'' & & \\ \text{inline}_S((\texttt{import}\ id\ c[: b]), I) & = & c : b, \text{inline}_S(I) \\ \quad \text{where } S = S', (\texttt{module}\ id\ I'\ X\ (B, c : b, B')), S'' & & \\ \text{inline}_S((\texttt{import}\ id\ c[: b]), I) & = & \text{inline}_{S'}(\texttt{import}\ id'\ c[: b]), \text{inline}_S(I) \\ \quad \text{where } S = S', (\texttt{module}\ id\ (I', \texttt{import}\ id'\ c[: b], I'')\ X\ B), S'' & & \end{cases}$$

A valid import list consists of either generic statements `import` $id\ *$, or of precise items `import` $id\ c[: b]$. In the first case, it is sufficient to verify that module id exists in the preceding part of the specification. In the second case, it is also necessary to verify that this module exports that item.

$S \vdash I$ imports	*I is a valid import list w.r.t. specification S*

$$\frac{}{S \vdash \cdot \text{ imports}}\ \textbf{vI\_}\cdot \qquad \frac{S \vdash I \text{ imports}}{\underbrace{S', (\texttt{module}\ id\ I'\ X\ B), S''}_{S} \vdash I, (\texttt{import}\ id\ *) \text{ imports}}\ \textbf{vI\_}*$$

$$\frac{S \vdash I \text{ imports} \qquad (c : b) \in (\text{inline}_{S'}(I'), B) \qquad \text{inline}_{S'}(I'), B \vdash c \text{ exports}}{\underbrace{S', (\texttt{module}\ id\ I'\ X\ B), S''}_{S} \vdash I, (\texttt{import}\ id\ c[: b]) \text{ imports}}\ \textbf{vI\_I}$$

This judgment relies on the judgment validating an export list w.r.t to an extended module ($B \vdash X$ exports), described next.

An extended body is always allowed to export $*$. An extended body exports a label c if an item $c : b$ is contained in it.

$B \vdash X$ exports	*X is a valid export list w.r.t. extended body B*

$$\frac{}{B \vdash \cdot \text{ exports}}\ \mathbf{vX\_\cdot}$$

$$\frac{B \vdash X \text{ exports}}{B \vdash X, (\texttt{export} *) \text{ exports}}\ \mathbf{vX\_*} \qquad \frac{B \vdash X \text{ exports}}{\underbrace{B', c : b, B''}_{B} \vdash X, (\texttt{export}\ c) \text{ exports}}\ \mathbf{vX\_I}$$

This judgment does not depend on any other judgment.

4.3.3 Bodies in Context

The (inlined) body of a module is a sequence of declarations, equations, definitions, and roles. Each of these objects can refer to identifiers declared or defined earlier in the body. The list of declared identifiers is maintained in a context.

Contexts

A *context* Γ is an ordered list of kind declarations, type declarations, or subsorting declarations. It is assumed that any declared symbol occurs exactly once in a context. A context is given by the following grammar:

$$\Gamma ::= \cdot \mid \Gamma, x : \kappa \mid \Gamma, x : \tau \mid \Gamma, x : \sigma$$

Note that contexts are not legal *MSR* entities, but they form a helper syntactic category for the purpose of type checking.

A context is valid if each declaration occurring in it is valid with respect to the earlier declarations in the context. Therefore, a kind declaration $x : \kappa$ is valid if κ is a valid kind with respect to the earlier part of this context. Similarly for type and subsort declarations.

$\vdash \Gamma$ context

$$\frac{}{\vdash \cdot \text{ context}}\ \mathbf{v\Gamma\_\cdot} \qquad \frac{\vdash \Gamma \text{ context} \quad \Gamma \vdash \kappa \text{ kind}}{\vdash \Gamma, x : \kappa \text{ context}}\ \mathbf{v\Gamma\_\kappa}$$

$$\frac{\vdash \Gamma \text{ context} \quad \Gamma \vdash \tau : \texttt{type}}{\vdash \Gamma, x : \tau \text{ context}}\ \mathbf{v\Gamma\_\tau} \qquad \frac{\vdash \Gamma \text{ context} \quad \Gamma \vdash \sigma \text{ subsort}}{\vdash \Gamma, x : \sigma \text{ context}}\ \mathbf{v\Gamma\_\sigma}$$

This judgment depends on the judgment validating a kind ($\Gamma \vdash \kappa$ kind), the judgment validating a type ($\Gamma \vdash \tau : \texttt{type}$), and the judgment validating a subsorting relation ($\Gamma \vdash \sigma$ subsort). All are described in the next section.

Bodies

A body is valid if all items it contains are valid in the context consisting at least of all preceding declarations.

$\Gamma \vdash B$ body	*B is a valid body in context Γ*

$$\frac{\vdash \Gamma \text{ context}}{\Gamma \vdash \cdot \text{ body}}\ \mathbf{vB\_\cdot} \qquad \frac{\Gamma \vdash \kappa \text{ kind} \quad \Gamma, x : \kappa \vdash B \text{ body}}{\Gamma \vdash x : \kappa, B \text{ body}}\ \mathbf{vB\_\kappa}$$

$$\frac{\Gamma \vdash \tau : \mathtt{type} \quad \Gamma, x : \tau \vdash B \text{ body}}{\Gamma \vdash x : \tau, B \text{ body}}\ \mathbf{vB\_\tau} \qquad \frac{\Gamma \vdash \sigma \text{ subsort} \quad \Gamma, x : \sigma \vdash B \text{ body}}{\Gamma \vdash x : \sigma, B \text{ body}}\ \mathbf{vB\_\sigma}$$

$$\frac{\Gamma \vdash \vec{o} \text{ body} \quad \Gamma, \vec{o} \vdash t : \tau \quad \Gamma, x : \{\vec{o}\}\tau \vdash B \text{ body}}{\Gamma \vdash x\ \vec{o} := t.\ B \text{ body}}\ \mathbf{vB\_D}$$

$$\frac{\Gamma \vdash E \text{ eq} \quad \Gamma \vdash B \text{ body}}{\Gamma \vdash x : E.\ B \text{ body}}\ \mathbf{vB\_E} \qquad \frac{\Gamma \vdash \rho \text{ role} \quad \Gamma \vdash B \text{ body}}{\Gamma \vdash x : \rho.\ B \text{ body}}\ \mathbf{vB\_\rho}$$

This judgment depends on the judgment that checks the validity of a context ($\vdash \Gamma$ context), defined above. It also depends on the judgments validating a kind ($\Gamma \vdash \kappa$ kind), a type ($\Gamma \vdash \tau : \mathtt{type}$), a subsorting relation ($\Gamma \vdash \sigma$ subsort), a term ($\Gamma \vdash t : \tau$), an equation ($\Gamma \vdash E$ eq) and a role ($\Gamma \vdash \rho$ role). The judgments in this last class will be defined in the next few sections.

Each of these rules adds an object to the current context. As they do so, they verify the validity of this object. Therefore, the check performed by rule **vB_·** can be omitted whenever this judgment is initially called with an empty context.

Rule **vB_D** validates a definition $x\ \vec{o} := t$ where x is the defined identifier, $\vec{o}$ is the list of parameters of the definition, and t is the defining term. The first premise verifies that each declaration in $\vec{o}$ is valid in the current context. With a slight abuse of notation, we rely on an independent call to the body validation judgment to perform this check. The second premise verifies that t is a valid term of some type τ in the current context extended with the parameters, the third premise makes x available for the validation of the rest of the body B. Its type is set to be the dependent type given by τ prefixed with each of the declarations in $\vec{o}$, in order. Formally, $\{\vec{o}\}\tau$ is defined as follows:

$$\begin{cases} \{\cdot\}\tau & = & \tau \\ \{(x : \tau')\vec{o}\}\tau & = & \{x : \tau\}(\{\vec{o}\}\tau) \end{cases}$$

The identifier prefixing a subsorting declaration and an equation is optional. When not present, it is assumed that the type checker synthesizes a new unique identifier and associates it with this entity.

4.3.4 Declarations

Declarations are issued for kinds, types and subsorts. This section defines the judgments that decide whether a kind, a type and a subsort are valid.

Kinds

Kinds classify types and type constructors. The kind `type` is valid in any context. A dependent kind is valid if each prefixing object declaration has a valid type with respect to the current context augmented with the object declarations preceding it.

$\Gamma \vdash \kappa$ kind	*κ is a valid kind in context Γ*

$$\frac{}{\Gamma \vdash \texttt{type}\ \mathsf{kind}}\ \mathbf{v}\kappa\_\texttt{type} \qquad \frac{\Gamma \vdash \tau : \texttt{type} \quad \Gamma, x : \tau \vdash \kappa\ \mathsf{kind}}{\Gamma \vdash \{x : \tau\}\kappa\ \mathsf{kind}}\ \mathbf{v}\kappa\_\mathbf{\Pi}$$

This judgment relies on the type validation judgment ($\Gamma \vdash \tau : \kappa$) defined later in this section.

Subsorts

A subsorting declaration is valid if both sides of $<:$ are valid types in the current context augmented with declarations for each prefixing dependent object.

$\Gamma \vdash \sigma$ subsort	*σ is a valid subsort in context Γ*

$$\frac{\Gamma \vdash \tau : \texttt{type} \quad \Gamma \vdash \tau' : \texttt{type}}{\Gamma \vdash \tau <: \tau'\ \mathsf{subsort}}\ \mathbf{v}\sigma\_<: \qquad \frac{\Gamma \vdash \tau : \texttt{type} \quad \Gamma, x : \tau \vdash \sigma\ \mathsf{subsort}}{\Gamma \vdash \{x : \tau\}\sigma\ \mathsf{subsort}}\ \mathbf{v}\sigma\_\mathbf{\Pi}$$

This judgment depends on the judgment validating types ($\Gamma \vdash \tau : \kappa$) defined next.

Subtypes

Type τ is a valid subtype of τ' if it is a valid subsort of τ', if both are obtained by instantiation with a valid term of the expected type, or if they can be linked transitively by these two rules.

$\Gamma \vdash \sigma$ inst	*σ is a valid subsort instance in context Γ*

$$\frac{\Gamma \vdash \sigma\ \mathsf{subsort}}{\Gamma, \sigma, \Gamma' \vdash \sigma\ \mathsf{inst}}\ \mathbf{vi}\sigma\_\mathit{base} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \{x : \tau\}\sigma\ \mathsf{inst}}{\Gamma \vdash [t/x]\sigma\ \mathsf{inst}}\ \mathbf{vi}\sigma\_\mathbf{\Pi}$$

$$\frac{\Gamma \vdash \tau <: \tau'\ \mathsf{inst} \quad \Gamma \vdash \tau' <: \tau''\ \mathsf{inst}}{\Gamma \vdash \tau <: \tau''\ \mathsf{inst}}\ \mathbf{vi}\sigma\_<:$$

This judgment depends on the just introduced judgment validating subsorts ($\Gamma \vdash \sigma$ subsort) and on the judgment validating terms ($\Gamma \vdash t : \tau$) defined in a section to come.

Types

Types classify terms. Types and type constructors are in turn classified by kinds. This judgment decides whether a type belongs to a given kind with respect to a given context. A constant declared in the context of some valid kind κ has that kind, unless it is the constant `state` that has kind `type` unconditionally. A base type $(a\ t)$ is validated by discovering the type τ of t and checking that a has a kind dependent on τ. The kind of $(a\ t)$ is obtained by replacing any mention x with t. A dependent type $\{x : \tau\}\tau'$ shall always have kind `type`, and the same applies for the embedded types τ and τ'. Because of the dependency on x, the type τ' must be validated in a context extended with the declaration $x : \tau$.

$\Gamma \vdash \tau : \kappa$	*τ is a valid type of kind κ in context Γ*

$$\frac{\Gamma \vdash \kappa \ \mathsf{kind}}{\Gamma, a : \kappa, \Gamma' \vdash a : \kappa}\ \mathbf{v}\tau\_base \qquad \frac{}{\Gamma \vdash \mathtt{state} : \mathtt{type}}\ \mathbf{vt\_state}$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash a : \{x : \tau\}\kappa}{\Gamma \vdash a\ t : [t/x]\kappa}\ \mathbf{v}\tau\_atm \qquad \frac{\Gamma \vdash \tau : \mathtt{type} \quad \Gamma, x : \tau \vdash \tau' : \mathtt{type}}{\Gamma \vdash \{x : \tau\}\tau' : \mathtt{type}}\ \mathbf{v}\tau\_\mathbf{\Pi}$$

This judgment relies on the judgment validating a kind ($\Gamma \vdash \kappa \ \mathsf{kind}$) defined above, on the judgment validating a term ($\Gamma \vdash t : \tau$) defined in a section to come, and on the substitution operation defined earlier.

4.3.5 Equations

An equation is valid if both sides have the same type in the current context augmented with declarations for each object in its `forall` prefix.

$\Gamma \vdash E \ \mathsf{eq}$	*E is a valid equation in context Γ*

$$\frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 = t_2 \ \mathsf{eq}}\ \mathbf{vE}\_= \qquad \frac{\Gamma \vdash \tau : \mathtt{type} \quad \Gamma, x : \tau \vdash E \ \mathsf{eq}}{\Gamma \vdash \mathtt{forall}\ x : \tau.\, E \ \mathsf{eq}}\ \mathbf{vE}\_\forall$$

This judgment depends on the judgment validating terms ($\Gamma \vdash t : \tau$) introduced in the next section and the judgment validating types ($\Gamma \vdash \tau : \kappa$) defined earlier.

4.3.6 Roles and Rules

Terms

A term is valid if it is a known constant applied to zero or more valid terms. An application $t\ t'$ is typechecked by discovering the type τ' of t' and checking that t a type $\{x : \tau'\}\tau$ dependent on τ'. The type of $t\ t'$ is obtained by replacing any mention of x in τ with t'. Rule $\mathbf{vt}\_ <:$ specifies that a term can also be validated by looking for any supertype. By rule $\mathbf{vt}\_ :$, a term can only be annotated with a valid type.

$\Gamma \vdash t : \tau$	*t is a valid term of type τ in context Γ*

$$\frac{\Gamma \vdash \tau : \mathtt{type}}{\Gamma, c : \tau, \Gamma' \vdash c : \tau}\ \mathbf{vt}\_base \qquad \frac{\Gamma \vdash t' : \tau' \quad \Gamma \vdash t : \{x : \tau'\}\tau}{\Gamma \vdash t\ t' : [t'/x]\tau}\ \mathbf{vt\_\Pi}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash (t : \tau) : \tau}\ \mathbf{vt}\_: \qquad \frac{\Gamma \vdash \tau <: \tau' \ \mathsf{inst} \quad \Gamma \vdash t : \tau'}{\Gamma \vdash t : \tau}\ \mathbf{vt}\_<:$$

This judgment depends on the judgment validating types ($\Gamma \vdash \tau : \kappa$) and on the judgment validating subtypes ($\Gamma \vdash \sigma \ \mathsf{inst}$), both introduced earlier.

Multisets

Multisets are possibly empty comma-separated lists of terms of type `state`.

$\Gamma \vdash m$ mset	*m is a valid multiset in context* Γ

$$\frac{}{\Gamma \vdash \texttt{empty}\ \mathsf{mset}}\ \mathbf{vm\_empty} \qquad \frac{\Gamma \vdash m\ \mathsf{mset} \quad \Gamma \vdash t : \texttt{state}}{\Gamma \vdash m, t\ \mathsf{mset}}\ \mathbf{vm\_,}$$

This judgment depends on the typing judgment for terms ($\Gamma \vdash t : \tau$) defined above.

Rules

A valid rule right-hand side is a multiset prefixed by zero or more variable declarations $\texttt{exists}\ x : \tau$, each for a valid type τ. The embedded multiset may make use of these variable declarations.

$\Gamma \vdash rhs$ rhs	*rhs is a valid right-hand side in context* Γ

$$\frac{\Gamma \vdash m\ \mathsf{mset}}{\Gamma \vdash m\ \mathsf{rhs}}\ \mathbf{vrhs\_}\mathit{base} \qquad \frac{\Gamma \vdash \tau : \texttt{type} \quad \Gamma, x : \tau \vdash rhs\ \mathsf{rhs}}{\Gamma \vdash \texttt{exists}\ x : \tau.\, rhs\ \mathsf{rhs}}\ \mathbf{vrhs\_}\exists$$

This judgment relies on the judgment validating types ($\Gamma \vdash \tau : \kappa$), defined earlier.

A valid rule has at its core a two valid multisets, the guard and the left-hand side, and a right-hand side. This core is prefixed by zero or more variable declarations $\texttt{forall}\ x : \tau$, each for a valid type τ. The constituents of core may make use of these variable declarations.

$\Gamma \vdash r$ rule	*r is a valid rule in context* Γ

$$\frac{\Gamma \vdash \tau : \texttt{type} \quad \Gamma, x : \tau \vdash r\ \mathsf{rule}}{\Gamma \vdash \texttt{forall}\ x : \tau.\, r\ \mathsf{rule}}\ \mathbf{vr\_}\forall \qquad \frac{\Gamma \vdash m\ \mathsf{mset} \quad \Gamma \vdash m'\ \mathsf{mset} \quad \Gamma \vdash rhs\ \mathsf{rhs}}{\Gamma \vdash m; m' => rhs\ \mathsf{rule}}\ \mathbf{vr\_=>}$$

This judgment depends on the judgments validating types ($\Gamma \vdash \tau : \kappa$), multisets ($\Gamma \vdash m$ mset), and right-hand sides ($\Gamma \vdash rhs$ rhs), all introduced earlier.

Roles

A rule sequence is valid it is either empty or a list of valid rules. Rules in a rule sequence may be interspersed with existential variable declarations $\texttt{exists}\ x : \tau$ which may be used in subsequent rules. The type τ of each such declaration shall be valid.

$\Gamma \vdash rs$ rules	*rs is a valid rule sequence in context* Γ

$$\frac{}{\Gamma \vdash \texttt{empty}\ \mathsf{rules}}\ \mathbf{vrs\_empty} \qquad \frac{\Gamma \vdash r\ \mathsf{rule} \quad \Gamma \vdash rs\ \mathsf{rules}}{\Gamma \vdash r.rs\ \mathsf{rules}}\ \mathbf{vrs\_.} \qquad \frac{\Gamma \vdash \tau : \texttt{type} \quad \Gamma, x : \tau \vdash rs\ \mathsf{rules}}{\Gamma \vdash \texttt{exists}\ x : \tau.\, rs\ \mathsf{rules}}\ \mathbf{vrs\_}\exists$$

This judgment depends on the previously defined judgments validating types ($\Gamma \vdash \tau : \kappa$) and rules ($\Gamma \vdash r$ rule).

A valid role is a valid rule sequence that is either prefixed by a variable declaration $\texttt{forall}\ x : \tau$ for a valid type τ, or paired with a declared constant x using the constructor $\texttt{for}\ x$.

$\Gamma \vdash \rho$ role	*ρ is a valid role in context* Γ

$$\frac{\Gamma \vdash \tau : \texttt{type} \quad \Gamma, x : \tau \vdash rs\ \mathsf{rules}}{\Gamma \vdash \texttt{forall}\ x : \tau.\, \{rs\}\ \mathsf{role}}\ \mathbf{v}\rho\mathbf{\_}\forall \qquad \frac{\Gamma \vdash rs\ \mathsf{rules}}{\underbrace{\Gamma', x : \tau, \Gamma''}_{\Gamma} \vdash \texttt{for}\ x.\, \{rs\}\ \mathsf{role}}\ \mathbf{v}\rho\mathbf{\_for}$$

This judgments depends on the previously defined judgments validating types ($\Gamma \vdash \tau : \kappa$) and rule sequences ($\Gamma \vdash rs$ rules).

4.4 Errors

An implementation of type checking for *MSR* shall accept any specification that can be given a typing derivation according to the rules presented in this chapter. Whenever an *MSR* specification cannot be validated according to these rules, the typechecker shall report an error. Error messages should be as precise as possible and contain enough information so that the programmer can quickly fix their cause.

Chapter 5

Execution

This chapter defines two variants of an abstract execution semantics for *MSR*. One is sequential, the other parallel. Both operate on fully reconstructed programs as described in Chapter 3. We also assume that module boundaries have been dissolved so that specifications can be seen as a single body. Neither semantics requires programs to be well typed. However, *MSR* is type safe under both, *i.e.,* execution of a well-typed program always transforms valid states into valid states.

While we do not discuss how to refine these abstract semantics into concrete *MSR* run-time systems in any detail, we briefly examine basic directives to allow users to control execution.

5.1 Snapshots

A state of execution, or *snapshot*, is a 4-tuple consisting of a multiset m of ground terms, a context Σ recording the symbols in use, an active role set R (defined next), and a multiset $\vec{E}$ of equations without free variables. We write this snapshot as $[m]^{R}_{\Sigma;\vec{E}}$ and abbreviate it as C when the components are unimportant.

$$C \ ::= \ [S]^{R}_{\Sigma;\vec{E}}$$

An *active role set* R is a multiset of active roles, where an *active role* rs^{A} is a rule sequence with distinguished owner A and without free variables. Active roles are partially instantiated role suffixes.

$$R \ ::= \ \cdot \ \mid \ R, rs^{\mathsf{A}}$$

The natural extension of the substitution operation is defined over active roles.

Both snapshots and active roles are run-time artifacts, unavailable to the programmer.

5.2 Execution Judgments

The two execution semantics of *MSR* examined here are specified on the basis of the following judgments. Their definition is given in the next section.

$t \gg_{\vec{E}} t'$	*Equality modulo equations*
$m \gg_{\vec{E}} m'$	*Left-hand side match*
$(rhs)_\Sigma \gg (m)_{\Sigma'}$	*Right-hand side instantiation*
$r \triangleright [m]_\Sigma \gg [m']_{\Sigma'}$	*Rule application*
$B \triangleright C \longrightarrow C'$	*One-step sequential firing*
$B \triangleright C \longrightarrow^* C'$	*Multi-step sequential firing*
$B \triangleright C \Longrightarrow C'$	*One-step parallel firing*
$B \triangleright C \Longrightarrow^* C'$	*Multi-step parallel firing*

5.3 Execution Rules

Both the sequential and the parallel execution semantics rely on the same notion of state equality and rule application.

5.3.1 Equational Matching

Ground terms t and t' are equal modulo ground equations $\vec{E}$ if rewriting some subterms of t using $\vec{E}$ yields t'.

$t \gg_{\vec{E}} t'$	*Equality modulo equations*

$$\frac{}{t \gg_{\vec{E}} t}\ \textbf{xeq\_id} \qquad \frac{}{t \gg_{(\vec{E}, t=t')} t'}\ \textbf{xeq\_eq} \qquad \frac{t_1 \gg_{\vec{E}} t_1' \quad t_2 \gg_{\vec{E}} t_2'}{t_1\, t_2 \gg_{\vec{E}} t_1'\, t_2'}\ \textbf{xeq\_app} \qquad \frac{t \gg_{\vec{E}} t'}{t : \tau \gg_{\vec{E}} t' : \tau}\ \textbf{xeq\_:}$$

States m and m' are equal modulo ground equations $\vec{E}$ if rewriting some subterms of m using $\vec{E}$ yields m'.

$m \gg_{\vec{E}} m'$	*Left-hand side match*

$$\frac{}{\texttt{empty} \gg_{\vec{E}} \texttt{empty}}\ \textbf{xeq\_empty} \qquad \frac{m \gg_{\vec{E}} m' \quad t \gg_{\vec{E}} t'}{m, t \gg_{\vec{E}} m', t'}\ \textbf{xeq\_,}$$

This judgment depends on the equality modulo equations judgment ($t \gg_{\vec{E}} t'$) just defined.

5.3.2 Rule Application

A right-hand side is instantiated by replacing each existential quantifier with a new symbol and recording it in the context.

$(rhs)_\Sigma \gg (m)_{\Sigma'}$	*Right-hand side instantiation*

$$\frac{}{(m)_\Sigma \gg (m)_\Sigma}\ \textbf{xr\_}\mathit{base} \qquad \frac{([\mathsf{a}/x]rhs)_{(\Sigma, \mathsf{a}:\tau)} \gg (m)_{\Sigma'}}{(\exists x : \tau.\, rhs)_\Sigma \gg (m)_{\Sigma'}}\ \textbf{xr\_}\exists$$

To apply a rule, we first instantiate each prefixed universal quantifier with a term of the appropriate type, obtaining a ground rule $m; m'_r \Rightarrow rhs$. This rule is *enabled* in a state m^* if m^* has the form $\bar{m}, m, m'$ where m'_r is equal to m' modulo the current set of equations. This rule instance is applied (or *fired*) in m^* by replacing the left-hand side m' with the result m'' of instantiating the right-hand side rhs. This yields the state $\bar{m}, m, m''$. Notice that the guard m must be present, but is not modified by firing this rule.

$r \triangleright [m]_{\Sigma;\vec{E}} \gg [m']_{\Sigma';\vec{E}}$	*Rule application*

$$\frac{\Sigma \vdash t : \tau \quad [t/x]r \triangleright [m]_{\Sigma;\vec{E}} \gg [m']_{\Sigma';\vec{E}}}{(\forall x : \tau.\, r) \triangleright [m]_{\Sigma;\vec{E}} \gg [m']_{\Sigma';\vec{E}}}\ \mathbf{xr\_\forall} \qquad \frac{m'_r \gg_{\vec{E}} m' \quad (rhs)_\Sigma \gg (m'')_{\Sigma'}}{(m; m'_r \Rightarrow rhs) \triangleright [\bar{m}, m, m']_{\Sigma;\vec{E}} \gg [\bar{m}, m, m'']_{\Sigma';\vec{E}}}\ \mathbf{xr\_\Rightarrow}$$

This judgment depends on the right-hand side instantiation judgment defined earlier.

5.3.3 Sequential Execution

Sequential execution, or *firing*, operates by copying a role to the active role set. Roles of the form $\texttt{forall}\, x : \tau.\, rs$ are first instantiated with a valid owner of type τ. Equations and definitions are copied to the current equation set and instantiated. Active roles are processed by instantiating existentials with appropriate constants which are recorded in the context, by applying a rule, or by skipping a rule. An empty rule sequence can be disposed of.

$B \triangleright C \longrightarrow C'$	*One-step sequential firing*

$$\frac{}{(B, \texttt{for}\ \mathsf{A}.\{rs\}) \triangleright [S]^{R}_{\Sigma;\vec{E}} \longrightarrow [S]^{R,rs^{\mathsf{A}}}_{\Sigma;\vec{E}}}\ \mathbf{xB\_for} \qquad \frac{\Sigma \vdash \mathsf{A} : \tau}{(B, \texttt{forall}\ x : \tau.\, \{rs\}) \triangleright [S]^{R}_{\Sigma;\vec{E}} \longrightarrow [S]^{R,([\mathsf{A}/x]rs)^{\mathsf{A}}}_{\Sigma;\vec{E}}}\ \mathbf{xB\_\forall}$$

$$\frac{}{B, (x\ \vec{o} := t) \triangleright [S]^{R}_{\Sigma;\vec{E}} \longrightarrow [S]^{R}_{\Sigma;(\vec{E}, \forall(\vec{o})\ x\ \vec{o} := t)}}\ \mathbf{xD}$$

$$\frac{}{(B, E) \triangleright [S]^{R}_{\Sigma;\vec{E}} \longrightarrow [S]^{R}_{\Sigma;(\vec{E}, E)}}\ \mathbf{xE} \qquad \frac{\Sigma \vdash t : \tau}{B \triangleright [S]^{R}_{\Sigma;(\vec{E}, \texttt{forall}\ x:\tau.\, E)} \longrightarrow [S]^{R}_{\Sigma;(\vec{E}, [t/x]E)}}\ \mathbf{xE\_\forall}$$

$$\frac{}{B \triangleright [S]^{R,(\texttt{exists}\ x:\tau.\, rs)^{\mathsf{A}}}_{\Sigma;\vec{E}} \longrightarrow [S]^{R,([\mathtt{c}/x]rs)^{\mathsf{A}}}_{(\Sigma,\mathtt{c}:\tau);\vec{E}}}\ \mathbf{xrs\_\exists} \qquad \frac{r \triangleright [S]_{\Sigma;\vec{E}} \gg [S']_{\Sigma';\vec{E}}}{B \triangleright [S]^{R,(r.rs)^{\mathsf{A}}}_{\Sigma;\vec{E}} \longrightarrow [S']^{R,(rs)^{\mathsf{A}}}_{\Sigma';\vec{E}}}\ \mathbf{xrs\_.}$$

$$\frac{}{B \triangleright [S]^{R,(r,rs)^{\mathsf{A}}}_{\Sigma;\vec{E}} \longrightarrow [S]^{R,(rs)^{\mathsf{A}}}_{\Sigma;\vec{E}}}\ \mathbf{xrs\_skp} \qquad \frac{}{B \triangleright [S]^{R,(\texttt{empty})^{\mathsf{A}}}_{\Sigma;\vec{E}} \longrightarrow [S]^{R}_{\Sigma;\vec{E}}}\ \mathbf{xrs\_empty}$$

Rule **xD** transforms a definition into an equation by prefixing it with a universal quantifier for each formal parameter. The notation $\forall(\vec{o})\ x\ \vec{o} := t$ is defined as follows:

$$\begin{cases} \forall(\cdot)\ x\ \vec{o} := t & = & x\ \vec{o} = t \\ \forall(x : \tau, \vec{o})\ D & = & \forall x : \tau.\, (\forall(\vec{o})\ D) \end{cases}$$

This judgment depends on the rule application judgment ($r \triangleright [S]_{\Sigma;\vec{E}} \gg [S']_{\Sigma'}$) defined earlier.

Multi-step sequential firing is defined as the reflexive and transitive closure of the one-step sequential firing judgment.

$B \triangleright C \longrightarrow^* C'$ *Multi-step sequential firing*

$$\frac{}{B \triangleright C \longrightarrow^* C}\ \mathbf{x*\_0} \qquad \frac{B \triangleright C \longrightarrow C' \quad B \triangleright C' \longrightarrow^* C''}{B \triangleright C \longrightarrow^* C''}\ \mathbf{x*\_n}$$

5.3.4 Parallel Execution

A single step of parallel execution is modeled by splitting the current snapshot into independent portions, doing one sequential firing step in each, and reassembling the result.

$B \triangleright C \Longrightarrow C'$ *One-step parallel firing*

$$\frac{}{B \triangleright C \Longrightarrow C}\ \mathbf{p\_id} \qquad \frac{B \triangleright [m_1]^{R_1}_{\Sigma;\vec{E}_1} \longrightarrow [m_1']^{R_1'}_{(\Sigma,\Sigma_1');\vec{E}_1'} \quad B \triangleright [m_2]^{R_2}_{\Sigma;\vec{E}_2} \Longrightarrow [m_2']^{R_2'}_{(\Sigma,\Sigma_2');\vec{E}_2'}}{B \triangleright [m_1, m_2]^{(R_1,R_2)}_{\Sigma;(\vec{E}_1,\vec{E}_2)} \Longrightarrow [m_1', m_2']^{(R_1',R_2')}_{(\Sigma,\Sigma_1',\Sigma_2');(\vec{E}_1',\vec{E}_2')}}\ \mathbf{p\_par}$$

This judgment depends on the single-step sequential execution judgment $B \triangleright C \longrightarrow^* C'$.

Multi-step parallel execution is the reflexive and transitive closure of single-step parallel execution

$B \triangleright C \Longrightarrow^* C'$ *Multi-step parallel firing*

$$\frac{}{B \triangleright C \Longrightarrow^* C}\ \mathbf{p*\_0} \qquad \frac{B \triangleright C \Longrightarrow C' \quad B \triangleright C' \Longrightarrow^* C''}{B \triangleright C \Longrightarrow^* C''}\ \mathbf{p*\_n}$$

5.4 Properties

MSR satisfies type preservation with respect to the typing semantics given in Chapter 4 and both execution semantics just examined.

Theorem 1 (Type preservation)

Let $\mathrm{decl}(B)$ *be the declarations occurring in* B*. Given snapshot* $[m]^{R}_{\Sigma;\vec{E}}$*, define* $B \vdash [m]^{R}_{\Sigma;\vec{E}}$ snapshot *to hold if* $\vdash \mathrm{decl}(B), \Sigma$ context *holds,* $\mathrm{decl}(B), \Sigma \vdash \vec{E}$ eq *holds,* $\mathrm{decl}(B), \Sigma \vdash m$ mset *holds, and* $\mathrm{decl}(B), \Sigma \vdash R$ role *holds.*

If $B \triangleright [m]^{R}_{\Sigma;\vec{E}} \longrightarrow [m']^{R'}_{\Sigma';\vec{E}'}$ *and* $\cdot \vdash B$ body *and* $B \vdash [S]^{R}_{\Sigma;\vec{E}}$ snapshot*, then* $B \vdash [S']^{R'}_{\Sigma';\vec{E}'}$ snapshot*.*

A detailed proof for a variant of this language can be found in [10]. Type preservation for multi-step and parallel firing are similar.

Type preservation ensures that, starting from a well-formed snapshot, execution of a well-typed specification cannot produce ill-formed snapshot. It also implies that execution can be implemented as not to rely on typing information except possibly for role owner instantiation, although this is a useful debugging tool.

5.5 Controlling Execution

A concrete run-time environment for *MSR* shall be correct with respect to the sequential or parallel semantics just presented. At the minimum, it shall give the user a command to run a specification from a specified

snapshot. However, this makes it hard to develop correct specifications and debug them in practice. It is recommended that the development environment provide commands to inspect the state, limit execution and control it. Some useful examples follow:

- `init` C: Set C as the current snapshot.
- `run` $[i]$ $[m]$: Execute the specification from the currents snapshot. If the argument i is present, do at most n steps. If the argument m is present, stop when a snapshot matches the parametric multiset m.
- `show`: Display the current snapshot.
- `choices`: Show the possibilities for the next execution step from the current snapshot.
- `choose` n: Choose the n-th execution step possibility from the current snapshot.

An implementation can make more sophisticated commands available [21].

5.6 Outputs

Type preservation ensures that running a well-typed *MSR* program cannot result in a run-time error.

In addition to the outputs of user commands intended to control execution (see above), it is useful to provide facilities for tracing and debugging an execution. This includes step-by-step snapshots, partial snapshots that focus on specific state elements, and execution statistics.

Bibliography

[1] Michael Backes, Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Cryptographically Sound Security Proofs for Basic and Public-Key Kerberos. *International Journal of Information Security — IJIS*, 10(2):107–134, 2011.

[2] Frederic Butler, Iliano Cervesato, Aaron D. Jaggard, and Andre Scedrov. A Formal Analysis of Some Properties of Kerberos 5 Using MSR. In *Fifteenth Computer Security Foundations Workshop — CSFW-15*, pages 175–190, Cape Breton, NS, Canada, 2002. IEEE Computer Society Press.

[3] Frederic Butler, Iliano Cervesato, Aaron D. Jaggard, and Andre Scedrov. Verifying Confidentiality and Authentication in Kerberos 5. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *Software Security — Theories and Systems — ISSS 2003*, pages 1–24, Tokyo, Japan, 2003. Springer-Verlag LNCS 3233.

[4] Frederic Butler, Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Christopher Walstad. Formal Analysis of Kerberos 5. *Theoretical Computer Science*, 367(1-2):57–87, 2006.

[5] Iliano Cervesato. Typed Multiset Rewriting Specifications of Security Protocols. In A. Seda, editor, *First Irish Conference on the Mathematical Foundations of Computer Science and Information Technology — MFCSIT'00*, pages 1–43, Cork, Ireland, 2000. Elsevier ENTCS 40.

[6] Iliano Cervesato. A Specification Language for Crypto-Protocols based on Multiset Rewriting, Dependent Types and Subsorting. In G. Delzanno, S. Etalle, and M. Gabbrielli, editors, *Workshop on Specification, Analysis and Validation for Emerging Technologies — SAVE'01*, pages 1–22, Paphos, Cyprus, 2001.

[7] Iliano Cervesato. Typed MSR: Syntax and Examples, First International Workshop on Mathematical Methods. In V.I. Gorodetski, V.A. Skormin, and L.J. Popyack, editors, *Models and Architectures for Computer Networks Security — MMM'01*, pages 159–177, St. Petersburg, Russia, 2001. Springer-Verlag LNCS 2052.

[8] Iliano Cervesato. Data Access Specification and the Most Powerful Symbolic Attacker in MSR. In M. Okada, B. Pierce, Andre Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security: Theories and Systems — ISSS 2002*, pages 384–416, Tokyo, Japan, 2002. Springer-Verlag LNCS 2609. Revised Papers of the 2002 Mext-NSF-JSPS International Symposium.

[9] Iliano Cervesato. Towards a Notion of Quantitative Security Analysis. In Dieter Gollmann, Fabio Massacci, and Artsiom Yautsiukhin, editors, *Quality of Protection: Security Measurements and Metrics — QoP'05*, pages 131–144. Springer-Verlag Advances in Information Security 23, 2006.

[10] Iliano Cervesato. Typed Multiset Rewriting Specifications of Security Protocols. Technical Report CMU-CS-11-140, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2011.

[11] Iliano Cervesato, Stefano Bistarelli, Gabriele Lenzini, Roberto Marangoni, and Fabio Martinelli. On Representing Biological Systems through Multiset Rewriting. In Roberto Moreno-Daz and Franz Pichler,

editors, *Nineth International Workshop on Computer Aided System Theory — EUROCAST'03*, pages 415–426, Las Palmas de Gran Canaria, Spain, 2003. Springer-Verlag LNCS 2809.

[12] Iliano Cervesato, Stefano Bistarelli, Gabriele Lenzini, and Fabio Martinelli. Relating Multiset Rewriting and Process Algebras for Security Protocol Analysis. *Journal of Computer Security*, 13(1):3–47, 2005.

[13] Iliano Cervesato, Nancy Durgin, Max I. Kanovich, and Andre Scedrov. Interpreting Strands in Linear Logic. In H. Veith, N. Heintze, and E. Clark, editors, *2000 Workshop on Formal Methods and Computer Security — FMCS'00*, Chicago, IL, 2000.

[14] Iliano Cervesato, Nancy Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Meta-Notation for Protocol Analysis. In *12th Computer Security Foundations Workshop — CSFW-12*, pages 55–69, Mordano, Italy, 1999. IEEE Computer Society Press.

[15] Iliano Cervesato, Nancy Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Comparison between Strand Spaces and Multiset Rewriting for Security Protocol Analysis. *Journal of Computer Security*, 13(2):265–316, 2005.

[16] Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Christopher Walstad. Specifying Kerberos 5 Cross-Realm Authentication. In Catherine Meadows and Jan Jürjens, editors, *Fifth Workshop on Issues in the Theory of Security — WITS'05*, pages 12–26, Long Beach, CA, 2005. ACM Digital Library.

[17] Iliano Cervesato, Aaron D. Jaggard, Joe-Kai Tsay, Andre Scedrov, and Christopher Walstad. Breaking and Fixing Public-Key Kerberos. *Information & Computation*, 206(2-4):402–424, 2008.

[18] Iliano Cervesato and Mark-Oliver Stehr. Representing the MSR Cryptoprotocol Specification Language in an Extension of Rewriting Logic with Dependent Types. *Higher-Order and Symbolic Computation*, 20(1/2):3–35, 2007.

[19] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12:247–311, 2004.

[20] Brigitte Pientka. An insider's look at LF type reconstruction: everything you (n)ever wanted to know. Submitted for publication, 2010.

[21] Stefan Reich. *User's Guide to the MSR Implementation.* University of Illinois, Urbana-Champaign, 2004.