SDL: A DSL for Cryptographic Schemes[1]

Leandro Facchinetti[2] *<leandro@jhu.edu>*

2017-11-14

[1] Research project report to fulfill a qualifying requirement of the Ph.D. program at The Johns Hopkins University.

[2] The author's work is supported by a CAPES fellowship. Process number: 13477/13-7.

IN THE PAST few years there has been an increasing interest in automating the development and implementation of cryptographic schemes, which traditionally are manual, difficult and error-prone tasks. Projects including AutoBatch and AutoGroup+ mechanically generate cryptographic schemes, and use a Domain-Specific Language (DSL) called the Scheme Description Language (SDL) to represent them; but, despite being a critical component in these systems, SDL is significantly limited. We introduce a redesigned SDL which addresses these shortcomings: we extend the language with new features, provide a formal specification, define a type system and several other static analyses, allow it to express various kinds of cryptographic schemes, and present techniques for manipulation via term rewriting. Finally, we explore two applications: a simple transformer to assert the consistent use of group operators (additive vs. multiplicative notation); and a generalization of AutoBatch, which includes more sophisticated term rewriting. Operations that previously required hundreds of lines of code in the original AutoBatch implementation are expressed with our techniques in less then ten lines. Overall, the redesigned SDL is a more solid and practical foundation upon which to build cryptographic-system development automations.

1 Introduction

CRYPTOGRAPHERS TRADITIONALLY develop cryptographic schemes manually, and communicate the results through papers. This work is difficult, prone to errors, and does not scale in accordance to the increasing demand for customized cryptographic schemes. But some tasks in the development of cryptographic schemes are mechanical, so recent research has been addressing these issues using automation.[3] For example, on the realm of pairing-based cryptography,[4] AutoBatch[5] mechanizes the development of verifiers for batches of digital signatures from schemes for a single signature, optimizing for parameters including the batch size. Along these lines, another project called AutoGroup+[6] automatically converts signature schemes using symmetric (Type-I) bilinear maps into signature schemes using asymmetric (Type-III) bilinear maps. The former are more convenient for humans, but the latter are more performant, and AutoGroup+ converts from one to the other while preserving the correctness and security proofs.

[3] Hoang et al. 2015, Malozemoff et al. 2014, Akinyele 2013.

[4] Koblitz and Menezes 2005.

[5] Akinyele et al. 2014a,b, Akinyele 2013.

[6] Akinyele et al. 2013b, Akinyele 2013, Akinyele et al. 2015.

Systems including AutoBatch and AutoGroup+ cannot directly work over specifications of cryptographic schemes on papers. Those descriptions are often too high-level, ambiguous and not machine-

friendly. On the other end of the abstraction spectrum there are implementations of these cryptographic schemes in general purpose programming languages (for example, C). These are not suitable for the purposes of automation either, for the opposite reason: they are too low-level and close to the machine, so the essence of the cryptographic schemes is lost in implementation details. We need a language with the right level of abstraction, that is as close as possible to the mathematical specifications cryptographers write on papers, but unambiguous, precise and machine-friendly.

The solution introduced by AutoBatch and AutoGroup+ is a Domain-Specific Language (DSL) called Scheme Description Language (SDL). SDL is not a general purpose programming language, it is constrained for the purpose of defining cryptographic schemes in a way that is convenient for cryptographers and machines alike.[7] SDL is a core component in AutoBatch and AutoGroup+, serving as input and output language, as well as internal representation for rewriting steps.

[7] Hudak 1997.

But the goal of AutoBatch and AutoGroup+ was not to design SDL, it was to explore automation in the development of pairing-based cryptographic schemes. And these projects were successful in that regard: AutoBatch, for example, was able to reproduce several results from the literature and discover novel optimizations. But, because the design of SDL was incidental, it has several flaws:

- According to the authors, the implementation is *research-grade software* in terms of code quality, and is not easy to maintain.
- There is no formal specification for the language.
- The language is artificially restricted to simplify the implementation. For example, iterations can only be nested one level deep.
- The language only has the features necessary to support pairing-based signature schemes and some encryption schemes such as identity-based encryption[8] and partial specification of attribute-based encryption.[9] However, the language does not fully support general cryptographic systems such as functional encryption and cryptographic protocols.
- There is a type system which detects basic errors cryptographers make when specifying schemes, but it is limited and requires a fair amount of manual annotation. And there are no other static analyses besides the type system.

[8] Boneh and Franklin 2001.

[9] Bethencourt et al. 2007.

WE INTRODUCE a redesigned SDL which addresses all the concerns above. We simplify the language to contain only the essential

features, while adding a few missing ones, for example, arbitrarily nested loops. Initially, we focus on pairing-based signature schemes, but the design is extensible for general cryptographic systems including public-key encryption, identity-based encryption, attribute-based encryption, cryptographic protocols and so forth. Our major contribution is a formal specification for SDL, which aids reasoning about cryptographic schemes both by machines and humans, including the following:

- A grammar for the abstract syntax (§ 3).[10]
- A collection of syntactic analyses to extract information from the cryptographic schemes (§ 4).
- A checker for well-formedness, which asserts proper use of identifiers and variable scope (§ 5).
- A type system with type inference (§ 6).
- A symmetry analysis (§ 7), to detect whether the cryptographic scheme uses symmetric (Type-I) bilinear maps or asymmetric (Type-III) bilinear maps, and to guarantee that they do not coalesce the two, which would be invalid.
- A dependence analysis (§ 8), to ascertain how data flows through the cryptographic scheme.
- A check of whether the cryptographic scheme is a valid signature scheme (§ 9). This includes checking for the presence of the necessary algorithms, that variables local to an algorithm are used, that information flows in a way to not disclose private information, and so forth.

[10] An abstract syntax is distinct from a concrete syntax for not being concerned with low-level details related to lexing and parsing, for example, the choice of characters or keywords for block delimiters.

BEYOND THE SDL specification, we also introduce techniques for manipulation via term rewriting applicable to cryptographic schemes (§ 10). They form an engine for mechanically exploring spaces of possible transformations over cryptographic schemes, with the possibility of optimizing for certain cost functions. We explore two applications:

- *Consistent use of group operators (§ 11)*. There are two notations for operations over group elements: multiplicative ($\times$ / $\div$) and additive ($+$ / $-$). While these notations are equivalent, it is necessary to consistently use only one or the other throughout the specification of a cryptographic scheme. Our system verifies this and, to facilitate further processing, converts from additive notation into multiplicative. This serves as a simple introductory example for the term-rewriting techniques.

- *AutoBatch generalization (§ 12)*. We recreate the core of the original AutoBatch system for the redesigned SDL. This requires more sophisticated term rewriting, including context-dependent rules. We only cover the core aspects of AutoBatch: semantics-preserving term rewriting and a goal-directed search. The rest of the system—for example, the heuristics dictated by cryptographers to prune the search space—is beyond the scope of this report.

SDL IS A specification DSL, decoupled from concrete implementations, but the end goal is to generate executable code for cryptographic schemes. A code-generation back-end can translate SDL into a library in a general-purpose programming language, and it is possible to develop multiple independent back-ends for different target languages using the term rewriting techniques presented above. So, to complete the specification of SDL, we define an interface expected by these back-ends for the primitive data and operations, for example, hash functions and exponentiation for group elements (§ 13).

ALL DEFINITIONS in SDL specification are executable code as well, because they were developed using a framework which is also at the core of our term-rewriting techniques: PLT Redex.[11] This document was typeset directly from the sources,[12] and serves as technical documentation for developers.

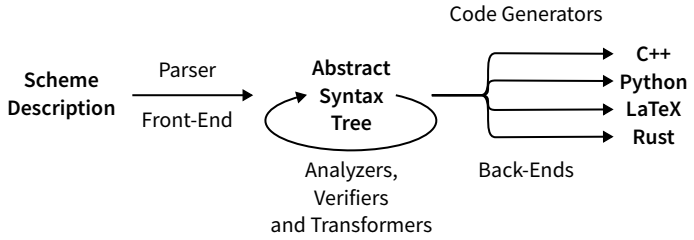
[11] Felleisen et al. 2009.

[12] URL `https://github.com/jakinyele/sdl-tools/tree/qualifying-project-report`

2 Architecture

THE HIGH-LEVEL SDL architecture is that of a traditional compiler:[13]

[13] Aho et al. 2007.



Cryptographers specify cryptographic schemes as plain text, which is parsed into an Abstract Syntax Tree (AST) by the compiler front-end. The AST also serves as an intermediate representation for a series of analyzers, verifiers and transformers. Analyzers extract information from the cryptographic schemes; verifiers find issues in the schemes and signal errors, preventing them from proceeding on the pipeline; and transformers rewrite the schemes, for example, to optimize them for a specific task. Finally, a back-end for code generation transforms the AST into a library in a general-purpose programming language.[14] As a special case, a LaTeX back-end typesets the crypto-

[14] The SDL compiler can be characterized as a **transpiler**, because it targets high-level programming languages, as opposed to low-level machine code.

graphic schemes, which is used for writing papers and inspecting the transformers.

THE SCOPE of this qualifying project is specifying the AST, and the analyzers, verifiers and transformers. *Specifically, the concrete syntax that cryptographers write, the parsing of this syntax, and code generation are beyond our scope (§ 14).* But we provide general techniques for term rewriting which can be used in code generators (§ 10), and we specify the interface for the cryptographic primitives in the generated code (§ 13). These cryptographic primitives are provided by third-party libraries for pairing-based cryptography, for example, Charm.[15]

[15] Akinyele et al. 2013a, Akinyele 2013.

ANALYZERS, VERIFIERS and transformers are defined as small, independent passes over the AST, to simplify the presentation and promote reuse. This is in the spirit of Nanopass,[16] though we do not use the tool. The passes are ordered and each of them assumes successful completion of the preceding, for example, the type inferencer (§ 6) is after the well-formedness verifier in the pipeline, so it is only defined for well-formed cryptographic schemes (§ 5). Moreover, later passes might depend on information collected by earlier passes. The following is the ordered list of passes:

[16] Sarkar et al. 2004, Keep 2013.

- Syntactic Analyses
- Well-Formedness Condition
- Type System—Type Inference
- Consistent Use of Group Operators
- Type System—Type Checking
- Symmetry Analysis
- Dependence Analysis
- Signature Scheme Verification
- Term Rewriting
- AutoBatch Generalization

Analyzer
Verifier
Transformer

WE USE REDEX to specify the AST, the passes, and the auxiliary data structures, for example, dependency graphs resulting from dependence analysis. The auxiliary data structures are represented as terms in an extended form of the core language. Analyzers and transformers are meta-functions, and verifiers are relations. The result is a runnable specification, which also serves as a reference implementation.

Several passes are divided in two parts following a naming convention: *transmogrify* is the high-level, more convenient definition that depends on *transmogrify**, which is the full low-level definition. Also, context-free meta-functions often employ a technique to recursively traverse the AST without having to restate all forms in the

language: the last two clauses match (**any** ...) and **any**, in order. This technique comes from the definition of *subst* in the Redex book.[17] We avoid using the unquote (**,**) form and *escaping back to Racket*, and keep as many definitions *in the language* as possible. The result is a standalone specification of SDL that does not depend on Racket, but auxiliary definition are necessary, for example, for operations on sets and graphs (§ A).

[17] Felleisen et al. 2009.

3 Abstract Syntax

WE INTRODUCE SDL'S abstract syntax by example. The following is the SDL specification for the BLS[18] cryptographic system:

[18] Boneh et al. 2004.

```
[(define-type M S)

 (define-algorithm (keygen)
   (define-output g (random G2))
   (define-local x (random ZR))
   (define-output pk (↑ g x))
   (define-output sk x))

 (define-algorithm (sign sk M)
   (define-output σ (↑ (H M G1) sk)))

 (define-algorithm (verify pk M σ g)
   (define-local/precomputed h (H M G1))
   (define-output valid? (= (E h pk) (E σ g))))]
```

The abstract syntax is represented by S-expressions, which are the results of parsing concrete syntax written by a cryptographer. The concrete syntax is more convenient for humans to write, because it has features including infix operators, for example. Both the concrete syntax and the corresponding parser are future work (§ 14).

A scheme in SDL is an ordered series of definitions (enclosed by square brackets—[]—in the example above). It is closer to a library than to a full program, because there are only declarations, and no actual computations to perform. In BLS, the first declaration is define-type, which introduces the name M (message) with type S (string). In general, SDL infers the types of expressions and variables, but there is information which comes from the outside world, for example, a message to sign, and it is mandatory to annotate its type. These are the only type annotations required in SDL.

The next definition is define-algorithm, which defines an algorithm that is part of the cryptographic scheme. The name of this first algorithm is keygen (key generation), and it has no inputs. The algorithm specification is a series of definitions, which are either define-local or define-output. The difference between the two is scope: variables defined with define-local are only in scope for the subsequent declara-

tions in the same algorithm; and variables defined with define-output, besides being in scope for the subsequent declarations in the same algorithm, are also available as potential inputs for the algorithms that follow.

After the definition keyword (define-local or define-output), there is the name of the defined variable, for example, g. Finally, there is the body of the definition, an expression for how to compute the variable. In the declarations for keygen we find expressions involving the primitives random and ↑. These primitives come with SDL and cover operations relevant to pairing-based encryption including hashing, arithmetic over group elements, bilinear maps and so forth. We treat these primitives as black boxes in SDL, and code generators expect libraries to provide them (§ 13). In our example, random draws a random element (from a uniform distribution) of the given type, G2 and ZR; and ↑ means exponentiation.

Variables in SDL are immutable, following the mathematical specifications generally found on papers. Also, their names must be unique throughout all algorithms in the scheme, a condition which is verified by the well-formedness checker (§ 5). An algorithm might have any number of outputs (define-output). The variable identifiers can include punctuation (for example, valid?), Greek letters (σ) and more.[19]

[19] The rules are the same as for Racket identifiers.

The second algorithm in BLS is sign, which has sk (secret key) and M (message) as inputs. SDL verifies the origin of the data coming in as inputs to an algorithm: sk is in scope because it is part of the output of a previous algorithm (keygen); and M comes from the outside world and was introduced by define-type. The definition of σ in sign is example of nested expressions in SDL, because the base of the exponentiation is (H M G1)—where H is a primitive for hashing the given value (M) into an element of the given type (G1). Operations including random and H might return values of various types, but these types are explicit on the specification, this allows for a complete type inferencer (§ 6).

The final algorithm in BLS, verify, includes a variation on define-local called define-local/precomputed. This is a hint to the code generator, which treats precomputed variables differently and promotes sharing to save computation. Also, the definition of valid? includes two new primitives: E is for bilinear maps[20] and = is mathematical equality. Operators including = and ↑ are overloaded and work over different types, for example, integers and group elements.

[20] In the literature, this operation is commonly represented with e.

THE SECOND EXAMPLE of a cryptographic system specified in SDL is CL[21]:

[21] Camenisch and Lysyanskaya 2004.

```
[(define-type M ZR)

 (define-algorithm (setup)
   (define-output g (random G1)))

 (define-algorithm (keygen g)
   (define-local x (random ZR))
   (define-local y (random ZR))
   (define-local X (↑ g x))
   (define-local Y (↑ g y))
   (define-output/composite pk (X Y))
   (define-output/composite sk (x y)))

 (define-algorithm (sign sk M)
   (define-local a (random G2))
   (define-local b (↑ a y))
   (define-local c (↑ a (+ x (× M x y))))
   (define-output/composite σ (a b c)))

 (define-algorithm (verify pk M σ g)
   (define-output valid?
     (and (= (E Y a) (E g b))
          (= (× (E X a) (↑ (E X b) M)) (E g c)))))]
```

The messages in CL are not strings (S), but elements of the ring of integers modulo r (ZR). And the specification of CL uses more primitives: and, × and +. These have the intuitive interpretation over booleans and integers.

More importantly, CL includes a new form of binding construct: define-output/composite. This defines an identifier by composing other identifiers, for example, pk (public key) is the composition of X and Y. When verify receives pk as input, the variables X and Y are in scope. In general, compositions can be part of other compositions, and all the composed identifiers become available when the composite binding is input to an algorithm. Intuitively, the composite bindings are records (or tuples) indexed by name, and these names become available without explicit projection.

THE FULL GRAMMAR for SDL's abstract syntax is the following:[22]

[22] The ellipses (...) mean "an arbitrary amount of the previous form."

```
          τ ::= S | I | B | ZR | G1 | G2 | GT | (L τ)
          s ::= [dt ... da ...]
         dt ::= (define-type x τ)
         da ::= (define-algorithm (x x/τ ...) de ...)
         de ::= (dk x/τ e) | (define-output/composite x (x/τ ...))
         dk ::= define-local | define-local/precomputed | define-output
          e ::= v | x | o
          v ::= string | integer | boolean
          o ::= (H e H/τ) | (random random/τ) | (init init/τ) | (op1 e) | (op2 e e) | (opn e e e ...) | (opl x e e e)
        H/τ ::= G1 | G2 | ZR
random/τ ::= S | I | B | ZR | G1 | G2 | GT
    init/τ ::= G1 | G2 | GT | ZR
       op1 ::= - | not | map
       op2 ::= E | ceillog | @ | @/I
       opn ::= = | + | - | × | ÷ | ↑ | and | or | ·
       opl ::= list | ∏ | ∑
       x/τ ::= x | [x : τ]
         x ::= variable-not-otherwise-mentioned
         p ::= ([x ↦ pv] ...)
        pv ::= string | integer | boolean | variable | (pv ...)
```

We start with the grammar for types (**τ**): strings (S), integers (I), booleans (B), integers modulo r (ZR), group elements (G1, G2 and GT), and lists (for example, lists of integers (L I)). The top-level forms in SDL are schemes (**s**), which are ordered sequences of definitions. Top-level definitions can be either types annotations for external data (**dt**) or algorithms (**da**). These definitions cannot be intertwined, all **dt**s must appear before the first **da**.

The definitions of algorithms expect their name (**x**) and a sequence of inputs (**x/τ**). The inputs are optionally annotated with a type ([**x** : **τ**]), in which case the type checker verifies the annotation agains the inferred type (§ 6). The body of algorithm definitions are ordered sequences of expression definitions (**de**). They come in two types: (**dk x/τ e**) and (define-output/composite **x** (**x/τ** ...)). The former is a regular variable definition, it starts with a definition keyword (**dk**), followed by a name with an optional type annotation (**x/τ**) and finally an expression (**e**). The definition keywords (**dk**) determine scope and inform the code generator of precomputed values. The other kind of expression definition (**de**) defines a composite binding, which behaves like a name-indexed tuple, as covered above. The parts that compose a binding of this form must be immediate identifiers with optional type annotations (**x/τ**), and arbitrary expressions are disallowed at this position.

Expressions (**e**) might be either literal value (**v**), variable references (**x**) or operations (**o**). Literal values in SDL are strings, integers and booleans. Variable references (**x**) are any name that does not appear elsewhere in the grammar. And operations (**o**) are the primitives in the language. The project's initial focus is on pairing-based encryption, so the primitives are bilinear maps, arithmetic on group

elements and so forth. But the set of primitives can be extended to other applications, and we anticipate either augmenting this grammar or designing a plugin mechanism that allow users to define their own primitives (§ 14). The following is a brief description of the primitives currently supported:

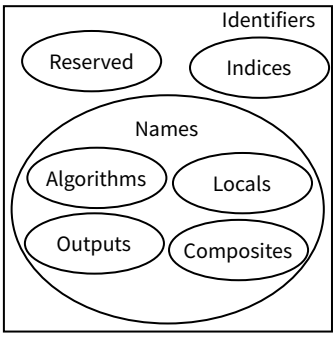
- H: Hashing of the result of **e** in the domain of **H/τ**. Not all types **τ** are allowed, and **H/τ** is the permitted subset.
- random: Draw a random element from the domain **random/τ**, which is also a subset of **τ**.
- init: Return an initial element from the domain **init/τ**, which is another subset of **τ**.
- - (unary), not, ceillog, =, +, - (binary), ×, ÷, ↑, and, or, ·: The intuitive interpretations of the arithmetic and boolean operators. - can be unary or binary. ↑ is exponentiation, and · is string concatenation. Mathematical operators are overloaded and work over different types of numbers, for example, integers and group elements. Equality = works over any operands of the same type. Operators of the **opn** non-terminal (for example, +) are variadic and accept two or more operands.
- map: Map strings to integers. To be used in conjunction with **@/I**, see below.
- E:[23] Bilinear maps. The inputs are either elements of G1 in the symmetric setting or one element of G1 and one element of G2 in the asymmetric setting (§ 7).
- @: List dereference. The inputs are, in order, the list and the integer index.
- @/I: Index into an integer. The inputs are, in order, the integer subject and the integer index. The result is also an integer. To be used in conjunction with map, see above.
- list: Create a list, with index **x** ranging from the result of the first given expression (**e**) to the result of the second given expression (inclusive), by repeatedly evaluating the third given expression. The index **x** is available in the subject expression (the last). Intuitively, this is similar to list comprehensions in Python, or to list unfolding in functional programming languages. Lists are immutable data structures.
- ∏ and ∑: Product and summation. The rules for indexing are the same as for list, see above. Intuitively, this is similar to list folding

[23] Represented by e in the literature.

(reducing) in functional programming languages, in which the subject is the list of indices.

TOOLS IN THE SDL ecosystem might need extra parameters besides the scheme definition, for example, AutoBatch optimizes the batch signature verification for a specific batch size. These parameters (**p**) are maps of names to values (**pv**), which can be strings, integers, booleans, symbols (**variable**), and lists of values.

THE SPACE of identifiers in SDL is partitioned:



- *Reserved*: Identifiers that appear in the grammar, for example, random and ↑.
- *Indices*: Variables that range over an interval in operations using **opl**. While in general variable names in SDL must be unique throughout the entire scheme definition, indices need no be unique. When nesting operations using **opl**, indices follow the rules of lexical (static) scoping.
- *Names*: Identifiers for algorithms (define-algorithm), general variable definitions (**dk**) and composites (define-output/composite). Names in SDL must be unique throughout the entire scheme definition.
- *Algorithms, locals and outputs*: Subsets of *names* that identify algorithms, locals and outputs, respectively.
- *Composites*: Subset of *names* for bindings which aggregate other bindings (define-output/composite).

4 Syntactic Analyses

WE INTRODUCE a series of syntactic analyses that extract information from cryptographic schemes with the purpose of simplifying the rest of the SDL ecosystem.

WE START by defining a language by extension of SDL, which contains forms for lists, sets, (multi-)maps and graphs:

X ::= (**x** ...)
g ::= ([**x** ↦ **X**] ...)

The **X** nonterminal refers to both lists and sets, and **g** refers to (multi-)maps and graphs, depending on how they are used. See § A for a collection of auxiliary operations over these data structures, including ∈, ⊆, and so forth for sets and *transitive-closure* for graphs.

We choose to keep these data structures and their operations *in the language*, so that the syntactical reasoning used in SDL applies to them as well. The system is constructive and self-contained.

THE *type-annotations* meta-function extracts type annotations (define-type) from cryptographic schemes:

type-annotations : **s** → ([**x** ↦ **τ**] ...)

type-annotations⟦[(define-type **x τ**) ... **da** ...]⟧ = ([**x** ↦ **τ**] ...)

The result of applying *type-annotations* to BLS (§ 3) is the following:

```
((M ↦ S))
```

And the result of applying *type-annotations* to CL (§ 3) is the following:

```
((M ↦ ZR))
```

THE *definitions/expressions* meta-function extracts all regular variable definitions (**dk**) from cryptographic schemes, indexed by their names:

definitions/expressions : _ → ([**x** ↦ **e**] ...)

definitions/expressions⟦(**dk x/τ e**)⟧ = ([*x*/*τ*->*x*⟦**x/τ**⟧ ↦ **e**])

definitions/expressions⟦(**any** ...)⟧ = ∪⟦*definitions/expressions*⟦**any**⟧, ...⟧

definitions/expressions⟦**any**⟧ = ()

The result of applying *definitions/expressions* to BLS (§ 3) is the following:

```
((g ↦ (random G2))
 (x ↦ (random ZR))
 (pk ↦ (↑ g x))
 (sk ↦ x)
 (σ ↦ (↑ (H M G1) sk))
 (h ↦ (H M G1))
 (valid? ↦ (= (E h pk) (E σ g))))
```

And the result of applying *definitions/expressions* to CL (§ 3) is the following:

```
((g ↦ (random G1))
 (x ↦ (random ZR))
 (y ↦ (random ZR))
 (X ↦ (↑ g x))
 (Y ↦ (↑ g y))
 (a ↦ (random G2))
 (b ↦ (↑ a y))
 (c ↦ (↑ a (+ x (× M x y))))
 (valid? ↦ (and (= (E Y a) (E g b)) (= (× (E X a) (↑ (E X b) M)) (E g c)))))
```

THE *definitions/composites* meta-function extracts all composite binding definitions (define-output/composite) from cryptographic schemes, indexed by their names:

definitions/composites : _ → **g**

definitions/composites⟦(define-output/composite **x** (**x/τ** ...))⟧ = ([**x** ↦ (*x*/*τ*-> *x*⟦**x/τ**⟧ ...)])

definitions/composites⟦(**any** ...)⟧ = ∪⟦*definitions/composites*⟦**any**⟧, ...⟧

definitions/composites⟦**any**⟧ = ()

The result of applying *definitions/composites* to BLS (§ 3) is the following:[24]

```
()
```

[24] BLS does not include composite bindings.

And the result of applying *definitions/composites* to CL (§ 3) is the following:

```
((pk ↦ (X Y)) (sk ↦ (x y)) (σ ↦ (a b c)))
```

THE *expressions* meta-function extracts all expressions from cryptographic schemes, even those nested in other expressions:

expressions : **s** → (**e** ...)

expressions⟦**s**⟧ = *expressions**⟦*ranges*⟦*definitions/expressions*⟦**s**⟧⟧⟧

*expressions** : _ → (**e** ...)

*expressions**⟦()⟧ = ()

*expressions**⟦**e**⟧ = ∪⟦(**e**), *expressions**⟦**any**⟧, ...⟧

where (**any** ...) = **e**

*expressions**⟦**e**⟧ = (**e**)

*expressions**⟦(**any** ...)⟧ = ∪⟦*expressions**⟦**any**⟧, ...⟧

*expressions**⟦**any**⟧ = ()

The result of applying *expressions* to BLS (§ 3) is the following:

```
((random G2)
 (random ZR)
 (↑ g x)
 g
 x
 (↑ (H M G1) sk)
 (H M G1)
 M
 sk
 (= (E h pk) (E σ g))
 (E h pk)
 h
 pk
 (E σ g)
 σ)
```

And the result of applying *expressions* to CL (§ 3) is the following:

```
((random G1)
 (random ZR)
 (↑ g x)
 g
 x
 (↑ g y)
 y
 (random G2)
 (↑ a y)
 a
 (↑ a (+ x (× M x y)))
 (+ x (× M x y))
 (× M x y)
 M
 (and (= (E Y a) (E g b)) (= (× (E X a) (↑ (E X b) M)) (E g c)))
 (= (E Y a) (E g b))
 (E Y a)
 Y
 (E g b)
 b
 (= (× (E X a) (↑ (E X b) M)) (E g c))
 (× (E X a) (↑ (E X b) M))
 (E X a)
 X
 (↑ (E X b) M)
 (E X b)
 (E g c)
 c)
```

THE *additions* + *subtractions* meta-function extracts all expressions which are additions or subtractions (+, -, ∑) from cryptographic schemes, even those nested in other expressions:

$$additions + subtractions : \mathbf{s} \to (\mathbf{e} \ldots)$$

$$additions + subtractions[\![\mathbf{s}]\!] = additions + subtractions^*[\![expressions[\![\mathbf{s}]\!]]\!]$$

$$additions + subtractions^* : (\mathbf{e} \ldots) \to (\mathbf{e} \ldots)$$

$$additions + subtractions^*[\![()]\!] = ()$$

$$additions + subtractions^*[\![(\mathbf{e}_1\ \mathbf{e}_3 \ldots)]\!] = \cup[\![(\mathbf{e}_1), additions + subtractions^*[\![(\mathbf{e}_3 \ldots)]\!]]\!]$$

where $\mathbf{e}_1 = (+\ \mathbf{e}_2 \ldots)$

$$additions + subtractions^*[\![(\mathbf{e}_1\ \mathbf{e}_3 \ldots)]\!] = \cup[\![(\mathbf{e}_1), additions + subtractions^*[\![(\mathbf{e}_3 \ldots)]\!]]\!]$$

where $\mathbf{e}_1 = (-\ \mathbf{e}_2 \ldots)$

$$additions + subtractions^*[\![(\mathbf{e}_1\ \mathbf{e}_3 \ldots)]\!] = \cup[\![(\mathbf{e}_1), additions + subtractions^*[\![(\mathbf{e}_3 \ldots)]\!]]\!]$$

where $\mathbf{e}_1 = (\sum\ \mathbf{any} \ldots)$

$$additions + subtractions^*[\![(\mathbf{any}\ \mathbf{e}_3 \ldots)]\!] = additions + subtractions^*[\![(\mathbf{e}_3 \ldots)]\!]$$

The result of applying *additions* + *subtractions* to BLS (§ 3) is the following:[25]

```
()
```

[25] BLS does not include additions or subtractions.

The result of applying *additions* + *subtractions* to CL (§ 3) is the following:

```
((+ x (× M x y)))
```

And the result of applying *additions + subtractions* to the artificial cryptographic scheme [(define-algorithm (keygen) (define-output sk (∑ z 1 10 2)))] is the following:

```
((∑ z 1 10 2))
```

THE *multiplications + divisions* meta-function extracts all expressions which are multiplications or divisions (×, ÷, ∏) from cryptographic schemes, even those nested in other expressions:

$$\begin{aligned}
&\textit{multiplications + divisions} : \mathbf{s} \to (\mathbf{e} \ldots)\\
&\textit{multiplications + divisions}[\![\mathbf{s}]\!] = \textit{multiplications + divisions}^*[\![\textit{expressions}[\![\mathbf{s}]\!]]\!]\\
&\textit{multiplications + divisions}^* : (\mathbf{e} \ldots) \to (\mathbf{e} \ldots)\\
&\textit{multiplications + divisions}^*[\![()]\!] = ()\\
&\textit{multiplications + divisions}^*[\![(\mathbf{e}_1\ \mathbf{e}_3 \ldots)]\!] = \cup[\![(\mathbf{e}_1), \textit{multiplications + divisions}^*[\![(\mathbf{e}_3 \ldots)]\!]]\!]\\
&\quad \text{where } \mathbf{e}_1 = (\times\ \mathbf{e}_2 \ldots)\\
&\textit{multiplications + divisions}^*[\![(\mathbf{e}_1\ \mathbf{e}_3 \ldots)]\!] = \cup[\![(\mathbf{e}_1), \textit{multiplications + divisions}^*[\![(\mathbf{e}_3 \ldots)]\!]]\!]\\
&\quad \text{where } \mathbf{e}_1 = (\div\ \mathbf{e}_2 \ldots)\\
&\textit{multiplications + divisions}^*[\![(\mathbf{e}_1\ \mathbf{e}_3 \ldots)]\!] = \cup[\![(\mathbf{e}_1), \textit{multiplications + divisions}^*[\![(\mathbf{e}_3 \ldots)]\!]]\!]\\
&\quad \text{where } \mathbf{e}_1 = (\textstyle\prod\ \mathbf{any} \ldots)\\
&\textit{multiplications + divisions}^*[\![(\mathbf{any}\ \mathbf{e}_3 \ldots)]\!] = \textit{multiplications + divisions}^*[\![(\mathbf{e}_3 \ldots)]\!]
\end{aligned}$$

The result of applying *multiplications + divisions* to BLS (§ 3) is the following:[26]

```
()
```

[26] BLS does not include multiplications or divisions.

The result of applying *multiplications + divisions* to CL (§ 3) is the following:

```
((× M x y) (× (E X a) (↑ (E X b) M)))
```

And the result of applying *multiplications + divisions* to the artificial cryptographic scheme [(define-algorithm (keygen) (define-output sk (∏ z 1 10 2)))] is the following:

```
((∏ z 1 10 2))
```

THE *type-assertions* meta-function extracts all type annotations (assertions for the type checker, § 6) from cryptographic schemes:

$$\begin{aligned}
&\textit{type-assertions} : _ \to ([\mathbf{x} \mapsto \boldsymbol{\tau}] \ldots)\\
&\textit{type-assertions}[\![[\mathbf{x} : \boldsymbol{\tau}]]\!] = ([\mathbf{x} \mapsto \boldsymbol{\tau}])\\
&\textit{type-assertions}[\![(\mathbf{any} \ldots)]\!] = \cup[\![\textit{type-assertions}[\![\mathbf{any}]\!], \ldots]\!]\\
&\textit{type-assertions}[\![\mathbf{any}]\!] = ()
\end{aligned}$$

The results of applying *type-assertions* to BLS and CL (§ 3) are the following:[27]

```
()
()
```

[27] Neither BLS nor CL include type assertions.

And the result of applying *multiplications + divisions* to the artificial cryptographic scheme [(define-algorithm (keygen [x : B]))] is the following:

((x ↦ B))

THE *inputs/s* meta-function extracts all inputs from cryptographic schemes, indexed by algorithm:[28]

[28] The *inputs/da* auxiliary meta-function is analogous, at the level of algorithms.

inputs/s : **s** → **g**

inputs/s⟦[**dt** ... **da** ...]⟧ = ([**x** ↦ *inputs/da*⟦**da**⟧] ...)
where **da** = (define-algorithm (**x x/τ** ...) **de** ...)

inputs/da : **da** → **X**

inputs/da⟦(define-algorithm (**x x/τ** ...) **de** ...)⟧ = (*x*/*τ*-> *x*⟦**x/τ**⟧ ...)

The result of applying *inputs/s* to BLS (§ 3) is the following:
((keygen ↦ ()) (sign ↦ (sk M)) (verify ↦ (pk M σ g)))

And the result of applying *inputs/s* to CL (§ 3) is the following:
((setup ↦ ()) (keygen ↦ (g)) (sign ↦ (sk M)) (verify ↦ (pk M σ g)))

THE *locals/s* meta-function extracts all locals (define-local and define-local/precomputed) from cryptographic schemes, indexed by algorithm:[29]

[29] The *locals/da* auxiliary meta-function is analogous, at the level of algorithms.

locals/s : **s** → **g**

locals/s⟦[**dt** ... **da** ...]⟧ = ([**x** ↦ *locals/da*⟦**da**⟧] ...)
where **da** = (define-algorithm (**x x/τ** ...) **de** ...)

locals/da : **da** → **X**

locals/da⟦(define-algorithm (**x x/τ**$_i$ ...)
(define-local **x/τ**$_o$ **e**) **de** ...)⟧ = ∪⟦(*x*/*τ*-> *x*⟦**x/τ**$_o$⟧),
locals/da⟦(define-algorithm (**x x/τ**$_i$ ...) **de** ...)⟧⟧

locals/da⟦(define-algorithm (**x x/τ**$_i$ ...)
(define-local/precomputed **x/τ**$_o$ **e**) **de** ...)⟧ = ∪⟦(*x*/*τ*-> *x*⟦**x/τ**$_o$⟧),
locals/da⟦(define-algorithm (**x x/τ**$_i$ ...) **de** ...)⟧⟧

locals/da⟦(define-algorithm (**x x/τ**$_i$ ...) **de**$_1$ **de**$_2$ ...)⟧ = *locals/da*⟦(define-algorithm (**x x/τ**$_i$ ...) **de**$_2$ ...)⟧

locals/da⟦(define-algorithm (**x x/τ** ...))⟧ = ()

The result of applying *locals/s* to BLS (§ 3) is the following:
((keygen ↦ (x)) (sign ↦ ()) (verify ↦ (h)))

And the result of applying *locals/s* to CL (§ 3) is the following:
((setup ↦ ()) (keygen ↦ (x y X Y)) (sign ↦ (a b c)) (verify ↦ ()))

THE *outputs/s* meta-function extracts all outputs (define-output and define-output/composite) from cryptographic schemes, indexed by algorithm:[30]

[30] The *outputs/da* auxiliary meta-function is analogous, at the level of algorithms.

outputs/s : **s** → **g**

outputs/s⟦[**dt** ... **da** ...]⟧ = ([**x** ↦ *outputs/da*⟦**da**⟧] ...)
where **da** = (define-algorithm (**x x/τ** ...) **de** ...)

outputs/da : **da** → **X**

outputs/da⟦(define-algorithm (**x x/τ**$_i$ ...) (define-output **x/τ**$_o$ **e**) **de** ...)⟧ = ∪⟦(*x*/*τ*- > *x*⟦**x/τ**$_o$⟧), *outputs/da*⟦(define-algorithm (**x x/τ**$_i$ ...) **de** ...)⟧⟧

outputs/da⟦(define-algorithm (**x**$_i$ **x/τ**$_i$ ...) (define-output/composite **x**$_c$ (**x/τ**$_c$ ...)) **de** ...)⟧ = ∪⟦(**x**$_c$), *outputs/da*⟦(define-algorithm (**x**$_i$ **x/τ**$_i$ ...) **de** ...)⟧⟧

outputs/da⟦(define-algorithm (**x x/τ**$_i$ ...) **de**$_1$ **de**$_2$ ...)⟧ = *outputs/da*⟦(define-algorithm (**x x/τ**$_i$ ...) **de**$_2$ ...)⟧

outputs/da⟦(define-algorithm (**x x/τ** ...))⟧ = ()

The result of applying *outputs/s* to BLS (§ 3) is the following:
((keygen ↦ (g pk sk)) (sign ↦ (σ)) (verify ↦ (valid?)))

And the result of applying *outputs/s* to CL (§ 3) is the following:
((setup ↦ (g)) (keygen ↦ (pk sk)) (sign ↦ (σ)) (verify ↦ (valid?)))

THE *names* meta-function extracts all names (algorithms, variables and composite bindings) from cryptographic schemes, including repetition:[31]

names : _ → **X**

names⟦[**dt** ... **da** ...]⟧ = ∪*⟦*names*⟦**dt**⟧, ..., *names*⟦**da**⟧, ...⟧

names⟦(define-type **x τ**)⟧ = *names*⟦**x**⟧

names⟦(define-algorithm (**x x/τ** ...) **de** ...)⟧ = ∪*⟦(**x**), *names*⟦**de**⟧, ...⟧

names⟦(**dk x/τ e**)⟧ = *names*⟦**x/τ**⟧

names⟦(define-output/composite **x** (**x/τ** ...))⟧ = *names*⟦**x**⟧

names⟦**x**⟧ = (**x**)

names⟦[**x** : **τ**]⟧ = *names*⟦**x**⟧

The result of applying *names* to BLS (§ 3) is the following:
(M keygen g x pk sk sign σ verify h valid?)

The result of applying *names* to CL (§ 3) is the following:
(M setup g keygen x y X Y pk sk sign a b c σ verify valid?)

And the result of applying *names* to the artificial cryptographic scheme [(define-algorithm (a) (define-local a 1))] is the following:
(a a)

[31] Repeating names is not allowed, and *names* is used to detect this issue (§ 5).

THE *algorithms* meta-function extracts all algorithm names (define-algorithm) from cryptographic schemes:[32]

algorithms : **s** → **X**

algorithms⟦[**dt** ... (define-algorithm (**x x/τ** ...) **de** ...) ...]⟧ = (**x** ...)

The result of applying *algorithms* to BLS (§ 3) is the following:
(keygen sign verify)

And the result of applying *algorithms* to CL (§ 3) is the following:
(setup keygen sign verify)

[32] The *algorithms* meta-function does not include repeated names, because it should only be applied to well-formed cryptographic schemes (§ 5).

THE *composites* meta-function extracts all composite bindings (define-output/composite) from cryptographic schemes:

composites : _ → **X**

composites⟦(define-output/composite **x** (**x/τ** ...))⟧ = (**x**)

composites⟦(**any** ...)⟧ = ∪⟦*composites*⟦**any**⟧, ...⟧

composites⟦**any**⟧ = ()

The result of applying *composites* to BLS (§ 3) is the following:

()

And the result of applying *composites* to CL (§ 3) is the following:

(pk sk σ)

THE *indices* meta-function extracts all indices (introduced by operations using **opl**) from cryptographic schemes:

indices : _ → **X**

indices⟦(**opl x** $\mathbf{e}_1$ $\mathbf{e}_2$ $\mathbf{e}_3$)⟧ = ∪⟦(**x**), *indices*⟦$\mathbf{e}_1$⟧, *indices*⟦$\mathbf{e}_2$⟧, *indices*⟦$\mathbf{e}_3$⟧⟧

indices⟦(**any** ...)⟧ = ∪⟦*indices*⟦**any**⟧, ...⟧

indices⟦**any**⟧ = ()

The result of applying *indices* to BLS and CL (§ 3) is the following:[33]

()

()

[33] Neither BLS nor CL include operations using **opl**.

And the result of applying *indices* to the artificial cryptographic scheme [(define-algorithm (keygen) (define-output sk (∑ x 1 10 20)))] is the following:

(x)

THE *free-indices* meta-function extracts all indices (introduced by operations using **opl**) which are free (not bound) in an expression:

free-indices : _ $\mathbf{X}_{\text{indices}}$ → **X**

free-indices⟦**x**, $\mathbf{X}_{\text{indices}}$⟧ = (**x**)

where ∈⟦**x**, $\mathbf{X}_{\text{indices}}$⟧

free-indices⟦**x**, $\mathbf{X}_{\text{indices}}$⟧ = ()

free-indices⟦(**opl x** $\mathbf{e}_1$ $\mathbf{e}_2$ $\mathbf{e}_3$), $\mathbf{X}_{\text{indices}}$⟧ = ∪⟦*free-indices*⟦$\mathbf{e}_1$, $\mathbf{X}_{\text{indices}}$⟧, *free-indices*⟦$\mathbf{e}_2$, $\mathbf{X}_{\text{indices}}$⟧, *set--*⟦*free-indices*⟦$\mathbf{e}_3$, $\mathbf{X}_{\text{indices}}$⟧, (**x**)⟧⟧

free-indices⟦(**any** ...), $\mathbf{X}_{\text{indices}}$⟧ = ∪⟦*free-indices*⟦**any**, $\mathbf{X}_{\text{indices}}$⟧, ...⟧

free-indices⟦**any**, $\mathbf{X}_{\text{indices}}$⟧ = ()

The result of applying *free-indices* to the expression (+ z x) when the set of indices is (z) is the following:

(z)

The result of applying *free-indices* to the expression (list x 1 10 x) when the set of indices is (x) is the following:

()

The result of applying *free-indices* to the expression (+ (∏ z 1 η z) x) when the set of indices is (z) is the following:

()

And the result of applying *free-indices* to the expression (+ (∏ z z z z) x) when the set of indices is (z) is the following:

(z)

5 Well-Formedness Condition

THE GRAMMAR FOR SDL (§ 3) allows for invalid cryptographic schemes, for example, a variable may be referenced before it is defined, or the a name may be reused. These are context-sensitive invariants, and checking for them is beyond the reach of context-free grammar we used to specify the abstract syntax. We introduce in this section a well-formedness checker that verifies the proper use of identifiers; more sophisticated passes, including a type system and semantic analyses are covered in the next sections.

A CRYPTOGRAPHIC SCHEME is well-formed if the following conditions are satisfied:

- All *names* are unique.
- The sets of *names* and *indices* are disjunct.
- All variables are defined before they are referenced.[34]

These conditions are checked in order, and each step assumes the previous was successful. Formally, the conditions are expressed by the following relations:

$$\frac{\textit{well-formed?}^*[\![\mathbf{s}, \textit{names}[\![\mathbf{s}]\!], \textit{indices}[\![\mathbf{s}]\!]]\!]}{\textit{well-formed?}[\![\mathbf{s}]\!]}$$

$$\frac{\textit{unique?}[\![\mathbf{X}_{\text{names}}]\!] \qquad \textit{disjunct?}[\![\mathbf{X}_{\text{names}}, \mathbf{X}_{\text{indices}}]\!] \qquad \textit{closed?}[\![\mathbf{s}]\!]}{\textit{well-formed?}^*[\![\mathbf{s}, \mathbf{X}_{\text{names}}, \mathbf{X}_{\text{indices}}]\!]}$$

The *well-formed?* relation is defined for schemes **s** and relies on auxiliary meta-functions to extract the relevant information from them: *names* and *indices* (§ 4). Then *well-formed?* relies on an auxiliary relation *well-formed?**, which verifies each of the invariants mentioned above via more auxiliary relations: *unique?* and *disjunct?*, which are just operations on lists and sets (§ A); and *closed?*, which is defined as follows:

$$\frac{\textit{closed?}^*[\![\mathbf{s}, (), (), \textit{transitive-closure}[\![\textit{definitions/composites}[\![\mathbf{s}]\!]]\!]]\!]}{\textit{closed?}[\![\mathbf{s}]\!]}$$

[34] If variables were referenced before their definition, then the meaning of the cryptographic scheme would be **open** to interpretation, up to the meaning of those variables. In SDL, we only consider schemes which do not include these variables, so are said to be **closed**.

$$\frac{\textit{closed?}^*[\![[\textbf{dt} \ldots\ \textbf{da} \ldots], \cup[\![\mathbf{X}_{\text{namespace/scheme}}, (\mathbf{x})]\!], (), \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]}{\textit{closed?}^*[\![[(\text{define-type } \mathbf{x}\ \boldsymbol{\tau})\ \textbf{dt} \ldots\ \textbf{da} \ldots], \mathbf{X}_{\text{namespace/scheme}}, (), \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]}$$

$$\frac{\begin{array}{c}\subseteq [\![\mathbf{X}_{\text{inputs}}, \mathbf{X}_{\text{namespace/scheme}}]\!] \\ \textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x})\ \textbf{de} \ldots)\ \textbf{da} \ldots], \mathbf{X}_{\text{namespace/scheme}}, \mathbf{X}_{\text{namespace/algorithm}}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!] \\ \mathbf{X}_{\text{inputs}} = (\mathbf{any}_1\ \mathbf{any}_2 \ldots) \\ \mathbf{X}_{\text{namespace/algorithm}} = \cup[\![(\mathbf{x}_{\text{input}} \ldots), \textit{lookup}^*[\![\mathbf{x}_{\text{input}}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, ()]\!], \ldots]\!] \\ (\mathbf{x}_{\text{input}} \ldots) = \mathbf{X}_{\text{inputs}} \\ \mathbf{X}_{\text{inputs}} = (x/\tau \text{-} > x[\![\mathbf{x}/\boldsymbol{\tau}]\!] \ldots)\end{array}}{\textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x}\ \mathbf{x}/\boldsymbol{\tau} \ldots)\ \textbf{de} \ldots)\ \textbf{da} \ldots], \mathbf{X}_{\text{namespace/scheme}}, (), \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]}$$

$$\frac{\begin{array}{c}\textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x})\ \textbf{de} \ldots)\ \textbf{da} \ldots], \\ \mathbf{X}_{\text{namespace/scheme}}, \cup[\![\mathbf{X}_{\text{namespace/algorithm}}, (x/\tau \text{-} > x[\![\mathbf{x}/\boldsymbol{\tau}]\!])]\!], \\ \textit{closed?}^*/e[\![\mathbf{e}, \mathbf{X}_{\text{namespace/algorithm}}]\!] \qquad \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]\end{array}}{\textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x})\ (\text{define-local } \mathbf{x}/\boldsymbol{\tau}\ \mathbf{e})\ \textbf{de} \ldots)\ \textbf{da} \ldots], \mathbf{X}_{\text{namespace/scheme}}, \mathbf{X}_{\text{namespace/algorithm}}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]}$$

$$\frac{\begin{array}{c}\textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x})\ \textbf{de} \ldots)\ \textbf{da} \ldots], \\ \mathbf{X}_{\text{namespace/scheme}}, \cup[\![\mathbf{X}_{\text{namespace/algorithm}}, (x/\tau \text{-} > x[\![\mathbf{x}/\boldsymbol{\tau}]\!])]\!], \\ \textit{closed?}^*/e[\![\mathbf{e}, \mathbf{X}_{\text{namespace/algorithm}}]\!] \qquad \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]\end{array}}{\textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x})\ (\text{define-local/precomputed } \mathbf{x}/\boldsymbol{\tau}\ \mathbf{e})\ \textbf{de} \ldots)\ \textbf{da} \ldots], \mathbf{X}_{\text{namespace/scheme}}, \mathbf{X}_{\text{namespace/algorithm}}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]}$$

$$\frac{\textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x}) \ \mathbf{de} \ldots) \ \mathbf{da} \ldots], \cup[\![\mathbf{X}_{\text{namespace/scheme}}, (x/\tau \text{-} > x[\![\mathbf{x}/\boldsymbol{\tau}]\!])]\!], \cup[\![\mathbf{X}_{\text{namespace/algorithm}}, (x/\tau \text{-} > x[\![\mathbf{x}/\boldsymbol{\tau}]\!])]\!], \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]}{\textit{closed?}^*/e[\![\mathbf{e}, \mathbf{X}_{\text{namespace/algorithm}}]\!]}$$

$$\textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x}) \ (\text{define-output } \mathbf{x}/\boldsymbol{\tau} \ \mathbf{e}) \ \mathbf{de} \ldots) \ \mathbf{da} \ldots], \mathbf{X}_{\text{namespace/scheme}}, \mathbf{X}_{\text{namespace/algorithm}}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]$$

$$\frac{\textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x}_{\text{algorithm}}) \ \mathbf{de} \ldots) \ \mathbf{da} \ldots], \cup[\![\mathbf{X}_{\text{namespace/scheme}}, (\mathbf{x}_{\text{composite}})]\!], \cup[\![\mathbf{X}_{\text{namespace/algorithm}}, (\mathbf{x}_{\text{composite}})]\!], \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]}{\textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x}_{\text{algorithm}}) \ (\text{define-output/composite } \mathbf{x}_{\text{composite}} \ (\mathbf{x}/\boldsymbol{\tau} \ldots)) \ \mathbf{de} \ldots) \ \mathbf{da} \ldots], \mathbf{X}_{\text{namespace/scheme}}, \mathbf{X}_{\text{namespace/algorithm}}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]}$$

$$\frac{\textit{closed?}^*[\![[\mathbf{da} \ldots], \mathbf{X}_{\text{namespace/scheme}}, (), \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]}{\textit{closed?}^*[\![[(\text{define-algorithm } (\mathbf{x})) \ \mathbf{da} \ldots], \mathbf{X}_{\text{namespace/scheme}}, \mathbf{X}_{\text{namespace/algorithm}}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]}$$

$$\textit{closed?}^*[\![[], \mathbf{X}_{\text{namespace/scheme}}, (), \mathbf{g}_{\text{definitions/composites/transitive-closure}}]\!]$$

$$\frac{}{\mathit{closed?}^*/e[\![\mathbf{v}, \mathbf{X}_{\text{namespace/algorithm}}]\!]}$$

$$\frac{\in[\![\mathbf{x}, \mathbf{X}_{\text{namespace/algorithm}}]\!]}{\mathit{closed?}^*/e[\![\mathbf{x}, \mathbf{X}_{\text{namespace/algorithm}}]\!]}$$

$$\frac{\mathit{closed?}^*/e[\![\mathbf{e}, \mathbf{X}_{\text{namespace/algorithm}}]\!]}{\mathit{closed?}^*/e[\![(\mathsf{H}\ \mathbf{e}\ \mathbf{H/\tau}), \mathbf{X}_{\text{namespace/algorithm}}]\!]}$$

$$\frac{}{\mathit{closed?}^*/e[\![(\mathsf{random}\ \mathbf{random/\tau}), \mathbf{X}_{\text{namespace/algorithm}}]\!]}$$

$$\frac{}{\mathit{closed?}^*/e[\![(\mathsf{init}\ \mathbf{init/\tau}), \mathbf{X}_{\text{namespace/algorithm}}]\!]}$$

$$\frac{\mathit{closed?}^*/e[\![\mathbf{e}, \mathbf{X}_{\text{namespace/algorithm}}]\!]}{\mathit{closed?}^*/e[\![(\mathbf{op1}\ \mathbf{e}), \mathbf{X}_{\text{namespace/algorithm}}]\!]}$$

$$\frac{\mathit{closed?}^*/e[\![\mathbf{e}_1, \mathbf{X}_{\text{namespace/algorithm}}]\!] \qquad \mathit{closed?}^*/e[\![\mathbf{e}_2, \mathbf{X}_{\text{namespace/algorithm}}]\!]}{\mathit{closed?}^*/e[\![(\mathbf{op2}\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{X}_{\text{namespace/algorithm}}]\!]}$$

$$\frac{\mathit{closed?}^*/e[\![\mathbf{e}, \mathbf{X}_{\text{namespace/algorithm}}]\!] \qquad \ldots}{\mathit{closed?}^*/e[\![(\mathbf{opn}\ \mathbf{e}\ \ldots), \mathbf{X}_{\text{namespace/algorithm}}]\!]}$$

$$\frac{\mathit{closed?}^*/e[\![\mathbf{e}_1, \mathbf{X}_{\text{namespace/algorithm}}]\!] \qquad \mathit{closed?}^*/e[\![\mathbf{e}_2, \mathbf{X}_{\text{namespace/algorithm}}]\!] \qquad \mathit{closed?}^*/e[\![\mathbf{e}_3, \cup[\![\mathbf{X}_{\text{namespace/algorithm}}, (\mathbf{x})]\!]]\!]}{\mathit{closed?}^*/e[\![(\mathbf{opl}\ \mathbf{x}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3), \mathbf{X}_{\text{namespace/algorithm}}]\!]}$$

The *closed?* relation depends on auxiliary meta-functions to extract information from the given scheme: *definitions/composites* (§ 4) and *transitive-closure* (§ A). We need the transitive closure of the graph of composite binding definitions because composite bindings might be part of other composite definitions. And, when one of them is input to an algorithm, all these bindings become available. Then *closed?* relation depends on the *closed?** auxiliary relation.

The *closed?** relation traverses the cryptographic scheme while carrying along two namespaces to determine scope: one for the entire scheme, and one local to an algorithm. The namespace for the entire scheme contains the variables bound by type declarations (define-type) and by outputs (define-output and define-output/composite) of preceding algorithms. This namespace is checked against algorithm inputs, data in a cryptographic system must either come from the outside world (in which case its type has been explicitly declared),

or it must be computed by the scheme itself (in which case it is the output of an algorithm and its type can be inferred; see § 6).

The other namespace is local to an algorithm. It is renewed when entering each algorithm on the traversal, and includes only the current algorithm inputs (along with their parts in the case of composite bindings) and accumulated definitions (either local or output). The expression (**e**) in each definition also has its scope checked with the auxiliary relation *closed?*/e*. Expressions using **opl** operators augment the namespace with the index they introduce.

6 *Type System*

SDL'S TYPE SYSTEM finds issues including incorrect uses of primitives, for example, adding an integer directly to a group element. Also, it checks the type annotations that cryptographers may add to definitions and algorithms inputs, either to improve their readability or to investigate bugs. The type system is divided in two parts: a type inferencer and a type checker. This allows transformers to rewrite schemes with knowledge of inferred types before they are checked. See § 11 for an example; it is a pass that rewrites schemes using additive notation for operations on group elements into schemes using multiplicative notation, which requires knowledge of types to find the additions and subtractions to convert. The advantage of having inference and checking as separate passes, and including transformation passes between them, is that the type checker can be more strict because the schemes have been normalized.

In this section we do not explore the type system's metatheory. We do not formally prove its soundness, for example. This would require a more rigorous notion of semantics and goes beyond the scope of this qualifying project (§ 14).

The type system requires an extended language:

τ ::= .... | ☠
Γ ::= ([**x** ↦ **τ**] ...)
G/τ ::= G1 | G2 | GT
+/-/τ ::= I | ZR
×/÷/τ ::= I | ZR | **G/τ**
↑/b/τ ::= I | ZR | **G/τ**
↑/e/τ ::= I | ZR

The first extension is on the notion of types itself (**τ**). We include the type ☠, which the type inferencer assigns to immediately inconsistent expressions and the type checker always considers a failure. We also introduce a typing environment (**Γ**), which maps identifiers (**x**) to their corresponding types (**τ**). The result of the type inferencer is a typing environment representing typing assumptions. These assumptions can be used by other passes before the type checking phase,

in particular to normalize schemes (§ 11). Then, the type checker verifies these assumptions.

The final language extension is a collection of nonterminals ***/τ**, which represent the types allowed by certain primitive operations. For example, **+/-/τ** states that only integers (I) and elements of the ring of integers modulo r (ZR) are allowed in additions or subtractions.[35]

[35] We assume that the additive notation for group operations has already been normalized into multiplicative notation (§ 11) at the point of checking types against the **+/-/τ** nonterminal, which explains the absence of **G/τ**.

THE TYPE INFERENCE meta-function *infer* receives as input a scheme **s** and returns a typing environment **Γ**, which represents a set of assumptions about the types for variables in the scheme:

$$
\begin{aligned}
&\mathit{infer} : \mathbf{s} \to \mathbf{\Gamma} \\
&\mathit{infer}[\![\mathbf{s}]\!] = \mathit{infer}^*[\![\mathit{definitions/expressions}[\![\mathbf{s}]\!], \\
&\qquad \cup[\![\mathit{type\text{-}annotations}[\![\mathbf{s}]\!], ([\mathbf{x}_{\text{indices}} \mapsto \mathsf{I}] \ldots), ([\mathbf{x}_{\text{composites}} \mapsto \text{🐱}] \ldots)]\!]] \\
&\text{where } (\mathbf{x}_{\text{indices}} \ldots) = \mathit{indices}[\![\mathbf{s}]\!], (\mathbf{x}_{\text{composites}} \ldots) = \mathit{composites}[\![\mathbf{s}]\!] \\
&\mathit{infer}^* : ([\mathbf{x} \mapsto \mathbf{e}] \ldots)\ \mathbf{\Gamma} \to \mathbf{\Gamma} \\
&\mathit{infer}^*[\![([\mathbf{x}_1 \mapsto \mathbf{e}_1]\ [\mathbf{x}_2 \mapsto \mathbf{e}_2] \ldots), \mathbf{\Gamma}]\!] = \mathit{infer}^*[\![([\mathbf{x}_2 \mapsto \mathbf{e}_2] \ldots), \cup[\![\mathbf{\Gamma}, ([\mathbf{x}_1 \mapsto \mathit{infer}^*/e[\![\mathbf{e}_1, \mathbf{\Gamma}]\!])]\!]]\!] \\
&\mathit{infer}^*[\![(), \mathbf{\Gamma}]\!] = \mathbf{\Gamma} \\
&\mathit{infer}^*/e : \mathbf{e}\ \mathbf{\Gamma} \to \boldsymbol{\tau} \\
&\mathit{infer}^*/e[\![\mathbf{string}, \mathbf{\Gamma}]\!] = \mathsf{S} \\
&\mathit{infer}^*/e[\![\mathbf{integer}, \mathbf{\Gamma}]\!] = \mathsf{I} \\
&\mathit{infer}^*/e[\![\mathbf{boolean}, \mathbf{\Gamma}]\!] = \mathsf{B} \\
&\mathit{infer}^*/e[\![\mathbf{x}, \mathbf{\Gamma}]\!] = \mathit{lookup}[\![\mathbf{x}, \mathbf{\Gamma}]\!] \\
&\mathit{infer}^*/e[\![(\mathsf{H}\ \mathbf{e}\ \mathbf{H/\tau}), \mathbf{\Gamma}]\!] = \mathbf{H/\tau} \\
&\mathit{infer}^*/e[\![(\mathsf{random}\ \mathbf{random/\tau}), \mathbf{\Gamma}]\!] = \mathbf{random/\tau} \\
&\mathit{infer}^*/e[\![(\mathsf{init}\ \mathbf{init/\tau}), \mathbf{\Gamma}]\!] = \mathbf{init/\tau} \\
&\mathit{infer}^*/e[\![(\text{-}\ \mathbf{e}), \mathbf{\Gamma}]\!] = \mathit{infer}^*/e[\![\mathbf{e}, \mathbf{\Gamma}]\!] \\
&\mathit{infer}^*/e[\![(\mathsf{not}\ \mathbf{e}), \mathbf{\Gamma}]\!] = \mathsf{B} \\
&\mathit{infer}^*/e[\![(\mathsf{map}\ \mathbf{e}), \mathbf{\Gamma}]\!] = \mathsf{I} \\
&\mathit{infer}^*/e[\![(\mathsf{E}\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{\Gamma}]\!] = \mathsf{GT} \\
&\mathit{infer}^*/e[\![(\mathsf{ceillog}\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{\Gamma}]\!] = \mathsf{I} \\
&\mathit{infer}^*/e[\![(@\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{\Gamma}]\!] = \boldsymbol{\tau} \\
&\text{where } (\mathsf{L}\ \boldsymbol{\tau}) = \mathit{infer}^*/e[\![\mathbf{e}_1, \mathbf{\Gamma}]\!] \\
&\mathit{infer}^*/e[\![(@\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{\Gamma}]\!] = \text{🐱} \\
&\mathit{infer}^*/e[\![(@/\mathsf{I}\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{\Gamma}]\!] = \mathsf{I} \\
&\mathit{infer}^*/e[\![(=\ \mathbf{e}_1\ \mathbf{e}_2 \ldots), \mathbf{\Gamma}]\!] = \mathsf{B} \\
&\mathit{infer}^*/e[\![(+\ \mathbf{e}_1\ \mathbf{e}_2 \ldots), \mathbf{\Gamma}]\!] = \mathit{infer}^*/e[\![\mathbf{e}_1, \mathbf{\Gamma}]\!] \\
&\mathit{infer}^*/e[\![(\text{-}\ \mathbf{e}_1\ \mathbf{e}_2 \ldots), \mathbf{\Gamma}]\!] = \mathit{infer}^*/e[\![\mathbf{e}_1, \mathbf{\Gamma}]\!] \\
&\mathit{infer}^*/e[\![(\times\ \mathbf{e}_1\ \mathbf{e}_2 \ldots), \mathbf{\Gamma}]\!] = \mathit{infer}^*/e[\![\mathbf{e}_1, \mathbf{\Gamma}]\!] \\
&\mathit{infer}^*/e[\![(\div\ \mathbf{e}_1\ \mathbf{e}_2 \ldots), \mathbf{\Gamma}]\!] = \mathit{infer}^*/e[\![\mathbf{e}_1, \mathbf{\Gamma}]\!] \\
&\mathit{infer}^*/e[\![(\uparrow\ \mathbf{e}_1\ \mathbf{e}_2 \ldots), \mathbf{\Gamma}]\!] = \mathit{infer}^*/e[\![\mathbf{e}_1, \mathbf{\Gamma}]\!] \\
&\mathit{infer}^*/e[\![(\mathsf{and}\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!] = \mathsf{B} \\
&\mathit{infer}^*/e[\![(\mathsf{or}\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!] = \mathsf{B} \\
&\mathit{infer}^*/e[\![(\cdot\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!] = \mathsf{S} \\
&\mathit{infer}^*/e[\![(\mathsf{list}\ \mathbf{x}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3), \mathbf{\Gamma}]\!] = (\mathsf{L}\ \mathit{infer}^*/e[\![\mathbf{e}_3, \mathbf{\Gamma}]\!]) \\
&\mathit{infer}^*/e[\![(\textstyle\prod\ \mathbf{x}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3), \mathbf{\Gamma}]\!] = \mathit{infer}^*/e[\![\mathbf{e}_3, \mathbf{\Gamma}]\!] \\
&\mathit{infer}^*/e[\![(\textstyle\sum\ \mathbf{x}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3), \mathbf{\Gamma}]\!] = \mathit{infer}^*/e[\![\mathbf{e}_3, \mathbf{\Gamma}]\!]
\end{aligned}
$$

The main meta-function *infer* deconstructs the scheme by using *definitions/expressions* (§ 4), and collects the type assumptions provided by the outside world: type annotations given by the cryptographer;

all indices, which have type integer (I); and composites, which are binding constructs that cannot be used in expression contexts, and therefore have type ☠.

Then, the auxiliary meta-function *infer** traverses the list of definitions in the scheme and calls the auxiliary meta-function *infer*/e* for each expression. The intermediary results are accumulated in the typing environment **Γ** and are available to subsequent definitions. Finally, the *infer*/e* meta-function receives an expression and a typing environment **Γ**, and produces a tentative type **τ**. Whether this type is correct will be determined by the type checker (see below), but it can be used for type-aware intermediary passes that only require type assumptions (§ 11).

The following are the type assumptions determined by the type inferencer for BLS:

((M ↦ S) (g ↦ G2) (x ↦ ZR) (pk ↦ G2) (sk ↦ ZR) (σ ↦ G1) (h ↦ G1) (valid? ↦ B))

And the following are the type assumptions determined by the type inferencer for CL:

((M ↦ ZR)
 (pk ↦ ☠)
 (sk ↦ ☠)
 (σ ↦ ☠)
 (g ↦ G1)
 (x ↦ ZR)
 (y ↦ ZR)
 (X ↦ G1)
 (Y ↦ G1)
 (a ↦ G2)
 (b ↦ G2)
 (c ↦ G2)
 (valid? ↦ B))

THE TYPE CHECKING relation *types-valid?* is defined for schemes **s** and determines whether the type assumptions from inference are consistent both with the definitions of the primitive operations and with the annotations given by the cryptographer:

$$\frac{\textit{types-valid?*}[\![\textit{type-assertions}[\![\mathbf{s}]\!], \textit{expressions}[\![\mathbf{s}]\!], \textit{infer}[\![\mathbf{s}]\!]]\!]}{\subseteq [\![\boldsymbol{\Gamma}_{\text{assertions}}, \boldsymbol{\Gamma}_{\text{inferred\_types}}]\!] \quad \textit{types-valid?}[\![\mathbf{s}]\!] \quad \textit{types-valid?*/e}[\![\mathbf{e}, \boldsymbol{\Gamma}_{\text{inferred\_types}}]\!] \quad \ldots}$$

$$\textit{types-valid?*}[\![\boldsymbol{\Gamma}_{\text{assertions}}, (\mathbf{e} \ldots), \boldsymbol{\Gamma}_{\text{inferred\_types}}]\!]$$

$$\frac{}{\textit{types-valid?*/e}[\![\mathbf{v}, \mathbf{\Gamma}]\!]}$$

$$\frac{}{\textit{types-valid?*/e}[\![\mathbf{x}, \mathbf{\Gamma}]\!]}$$

$$\frac{}{\textit{types-valid?*/e}[\![(\mathsf{H}\ \mathbf{e}\ \mathbf{H/\tau}), \mathbf{\Gamma}]\!]}$$

$$\frac{}{\textit{types-valid?*/e}[\![(\mathsf{random}\ \mathbf{random/\tau}), \mathbf{\Gamma}]\!]}$$

$$\frac{}{\textit{types-valid?*/e}[\![(\mathsf{init}\ \mathbf{init/\tau}), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathbf{+/-/\tau} = \textit{infer*/e}[\![\mathbf{e}, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(\mathsf{-}\ \mathbf{e}), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathsf{B} = \textit{infer*/e}[\![\mathbf{e}, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(\mathsf{not}\ \mathbf{e}), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathsf{S} = \textit{infer*/e}[\![\mathbf{e}, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(\mathsf{map}\ \mathbf{e}), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathsf{G1} = \textit{infer*/e}[\![\mathbf{e}_2, \mathbf{\Gamma}]\!] \qquad \mathsf{G1} = \textit{infer*/e}[\![\mathbf{e}_1, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(\mathsf{E}\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathsf{G2} = \textit{infer*/e}[\![\mathbf{e}_2, \mathbf{\Gamma}]\!] \qquad \mathsf{G1} = \textit{infer*/e}[\![\mathbf{e}_1, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(\mathsf{E}\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_2, \mathbf{\Gamma}]\!] \qquad \mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_1, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(\mathsf{ceillog}\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_2, \mathbf{\Gamma}]\!] \qquad (\mathsf{L}\ \boldsymbol{\tau}) = \textit{infer*/e}[\![\mathbf{e}_1, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(@\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_2, \mathbf{\Gamma}]\!] \qquad \mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_1, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(@/\mathsf{I}\ \mathbf{e}_1\ \mathbf{e}_2), \mathbf{\Gamma}]\!]}$$

$$\frac{\textit{all} = \textit{?}[\![(\textit{infer*/e}[\![\mathbf{e}, \mathbf{\Gamma}]\!] \ldots)]\!]}{\textit{types-valid?*/e}[\![(=\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!]}$$

$$\frac{\textit{all} = \textit{?}[\![(\mathbf{+/-/\tau} \ldots)]\!] \qquad (\mathbf{+/-/\tau} \ldots) = (\textit{infer*/e}[\![\mathbf{e}, \mathbf{\Gamma}]\!] \ldots)}{\textit{types-valid?*/e}[\![(+\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!]}$$

$$\frac{\textit{all} = \textit{?}[\![(\mathbf{+/-/\tau} \ldots)]\!] \qquad (\mathbf{+/-/\tau} \ldots) = (\textit{infer*/e}[\![\mathbf{e}, \mathbf{\Gamma}]\!] \ldots)}{\textit{types-valid?*/e}[\![(-\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!]}$$

$$\frac{\textit{all} = \textit{?}[\![(\mathbf{\times/\div/\tau} \ldots)]\!] \qquad (\mathbf{\times/\div/\tau} \ldots) = (\textit{infer*/e}[\![\mathbf{e}, \mathbf{\Gamma}]\!] \ldots)}{\textit{types-valid?*/e}[\![(\times\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!]}$$

$$\frac{\textit{all} = \textit{?}[\![(\mathbf{\times/\div/\tau} \ldots)]\!] \qquad (\mathbf{\times/\div/\tau} \ldots) = (\textit{infer*/e}[\![\mathbf{e}, \mathbf{\Gamma}]\!] \ldots)}{\textit{types-valid?*/e}[\![(\div\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!]}$$

$$\frac{(\mathbf{\uparrow/e/\tau} \ldots) = (\textit{infer*/e}[\![\mathbf{e}_e, \mathbf{\Gamma}]\!] \ldots) \qquad \mathbf{\uparrow/b/\tau} = \textit{infer*/e}[\![\mathbf{e}_b, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(\uparrow\ \mathbf{e}_b\ \mathbf{e}_e \ldots), \mathbf{\Gamma}]\!]}$$

$$\frac{(\mathsf{B} \ldots) = (\textit{infer*/e}[\![\mathbf{e}, \mathbf{\Gamma}]\!] \ldots)}{\textit{types-valid?*/e}[\![(\mathsf{and}\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!]}$$

$$\frac{(\mathsf{B} \ldots) = (\textit{infer*/e}[\![\mathbf{e}, \mathbf{\Gamma}]\!] \ldots)}{\textit{types-valid?*/e}[\![(\mathsf{or}\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!]}$$

$$\frac{}{\textit{types-valid?*/e}[\![(\cdot\ \mathbf{e} \ldots), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_2, \mathbf{\Gamma}]\!] \qquad \mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_1, \mathbf{\Gamma}]\!] \qquad \mathsf{I} = \textit{infer*/e}[\![\mathbf{x}, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(\mathsf{list}\ \mathbf{x}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathbf{\times/\div/\tau} = \textit{infer*/e}[\![\mathbf{e}_3, \mathbf{\Gamma}]\!] \qquad \mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_2, \mathbf{\Gamma}]\!] \qquad \mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_1, \mathbf{\Gamma}]\!] \qquad \mathsf{I} = \textit{infer*/e}[\![\mathbf{x}, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(\prod\ \mathbf{x}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3), \mathbf{\Gamma}]\!]}$$

$$\frac{\mathbf{+/-/\tau} = \textit{infer*/e}[\![\mathbf{e}_3, \mathbf{\Gamma}]\!] \qquad \mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_2, \mathbf{\Gamma}]\!] \qquad \mathsf{I} = \textit{infer*/e}[\![\mathbf{e}_1, \mathbf{\Gamma}]\!] \qquad \mathsf{I} = \textit{infer*/e}[\![\mathbf{x}, \mathbf{\Gamma}]\!]}{\textit{types-valid?*/e}[\![(\sum\ \mathbf{x}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3), \mathbf{\Gamma}]\!]}$$

The structure of the type checker *types-valid?* is similar to the

structure of the type inferencer *infer*: it extracts information from the scheme by delegating to other meta-functions, and then calls auxiliary meta-functions. These auxiliary meta-functions verify that the type assertions written by the cryptographer in annotations are consistent with the inferred types (using $\subseteq$), and that the primitive operations are used properly. The *types-valid?*/e* relation does not need to recursively call itself with subexpressions because the *expressions* meta-function (§ 4) collected all of them.

7 Symmetry Analysis

PAIRING-BASED CRYPTOGRAPHIC SCHEMES exist in two settings: symmetric (Type-I) and asymmetric (Type-III). Symmetric schemes are more convenient for cryptographers, and, consequently, are the variant more commonly found in papers. But asymmetric schemes are more performant, so implementers prefer them.[36] The difference between the settings lies on the use of bilinear maps (generally represented by e in the literature and by `E` in SDL): in symmetric schemes, both operands are of type `G1`; while in asymmetric schemes the operands have the types `G1` and `G2`. In either case the result is in `GT`.[37]

Symmetric and asymmetric uses of bilinear maps are disallowed within a single cryptographic scheme. We introduce relations that hold when a given scheme is symmetric, or when it is asymmetric; and another relation holds when the scheme makes consistent use of only a single symmetry.

THE *symmetric?* relation holds when the cryptographic scheme is symmetric:

$$\frac{\mathit{symmetric?}^*[\![\mathit{expressions}^*[\![\mathbf{s}]\!], \mathit{infer}[\![\mathbf{s}]\!]]\!] \qquad \mathtt{G1} = \mathit{infer}^*/e[\![\mathbf{e}_3, \boldsymbol{\Gamma}]\!]}{\mathit{symmetric?}[\![\mathbf{s}]\!] \qquad \mathit{symmetric?}^*[\![(\mathbf{e}_1 \ldots (\mathtt{E}\ \mathbf{e}_2\ \mathbf{e}_3)\ \mathbf{e}_4 \ldots), \boldsymbol{\Gamma}]\!]}$$

The definition of *symmetric?* depends on auxiliary meta-functions *expressions** (§ 4) and *infer* (§ 6) to extract the relevant information from the scheme. Then it uses the auxiliary relation *symmetric?**, which holds when there exists a bilinear map (`E`) in the scheme such that its second input is of type `G1`.

THE *asymmetric?* relation holds when the cryptographic scheme is asymmetric:

$$\frac{\mathit{asymmetric?}^*[\![\mathit{expressions}^*[\![\mathbf{s}]\!], \mathit{infer}[\![\mathbf{s}]\!]]\!] \qquad \mathtt{G2} = \mathit{infer}^*/e[\![\mathbf{e}_3, \boldsymbol{\Gamma}]\!]}{\mathit{asymmetric?}[\![\mathbf{s}]\!] \qquad \mathit{asymmetric?}^*[\![(\mathbf{e}_1 \ldots (\mathtt{E}\ \mathbf{e}_2\ \mathbf{e}_3)\ \mathbf{e}_4 \ldots), \boldsymbol{\Gamma}]\!]}$$

These definitions are analogous to those of *symmetric?*, except that it holds when there exists a bilinear map (`E`) in the scheme such that

[36] One SDL use case we contemplate is automatically converting symmetric into asymmetric schemes, improving on AutoGroup+ (§ 14).

[37] In summary:
Symmetric (Type-I)
`E : G1 G1 → GT`
Asymmetric (Type-III)
`E : G1 G2 → GT`
In theory, the order of the inputs in the asymmetric setting (G1 G2 vs. G2 G1) is arbitrary—as long as it is consistent throughout the scheme. In SDL we make the simplifying assumption that cryptographers always write inputs in the order G1 G2.

its second input is of type G2. Both BLS and CL, as defined in § 3, are asymmetric, so only *symmetric?* holds for them.

FINALLY, WE DEFINE the relation *consistent-symmetry?*, which depends on the relations above and holds when the given cryptographic scheme does not include both symmetric and asymmetric bilinear maps:

$$\frac{\mathit{consistent\text{-}symmetry?}^*[\![\mathit{expressions}^*[\![\mathbf{s}]\!], \mathit{infer}[\![\mathbf{s}]\!]]\!] \quad \neg[\![\wedge[\![\mathit{symmetric?}^*[\![\mathbf{any}, \Gamma]\!], \mathit{asymmetric?}^*[\![\mathbf{any}, \Gamma]\!]]\!]]}{\mathit{consistent\text{-}symmetry?}[\![\mathbf{s}]\!] \qquad \mathit{consistent\text{-}symmetry?}^*[\![\mathbf{any}, \Gamma]\!]}$$

8 Dependence Analysis

VARIABLE DEFINITIONS AND USES determine a flow of data within a cryptographic scheme, and, consequently, security properties. For example, in a signature scheme, the signature must depend on the private key and on the message; and verifying it must depend on the public key and on the message, but not on the private key. We introduce a dependence analysis which produces a dependency graph, which supports the checks for these invariants (§ 9). Moreover, the dependency graph informs sophisticated term-rewriting techniques, including AutoGroup+, which we will explore in the future (§ 14).

THE *dependencies* meta-function receives as input a cryptographic scheme and outputs its dependency graph:

$\mathit{dependencies} : \mathbf{s} \to \mathbf{g}$

$\mathit{dependencies}[\![\mathbf{s}]\!] = \mathit{dependencies}^*[\![\mathbf{s}, \mathit{indices}[\![\mathbf{s}]\!]]\!]$

$\mathit{dependencies}^* : \_\ \mathbf{X}_{\mathit{indices}} \to \mathbf{g}$

$\mathit{dependencies}^*[\![[\mathbf{dt} \ldots\ \mathbf{da} \ldots], \mathbf{X}_{\mathit{indices}}]\!] = \cup[\![\mathit{dependencies}^*[\![\mathbf{dt}, \mathbf{X}_{\mathit{indices}}]\!], \ldots, \mathit{dependencies}^*[\![\mathbf{da}, \mathbf{X}_{\mathit{indices}}]\!], \ldots]$

$\mathit{dependencies}^*[\![(\text{define-type } \mathbf{x}\ \boldsymbol{\tau}), \mathbf{X}_{\mathit{indices}}]\!] = ([\mathbf{x} \mapsto ()])$

$\mathit{dependencies}^*[\![(\text{define-algorithm } (\mathbf{x}\ \mathbf{x}/\boldsymbol{\tau} \ldots)\ \mathbf{de} \ldots), \mathbf{X}_{\mathit{indices}}]\!] = \cup[\![\mathit{dependencies}^*[\![\mathbf{de}, \mathbf{X}_{\mathit{indices}}]\!], \ldots]$

$\mathit{dependencies}^*[\![(\mathbf{dk}\ \mathbf{x}/\boldsymbol{\tau}\ \mathbf{e}), \mathbf{X}_{\mathit{indices}}]\!] = ([x/\tau\text{->}x[\![\mathbf{x}/\boldsymbol{\tau}]\!] \mapsto \mathit{dependencies/e}[\![\mathbf{e}, \mathbf{X}_{\mathit{indices}}]\!]])$

$\mathit{dependencies}^*[\![(\text{define-output/composite } \mathbf{x}\ (\mathbf{x}/\boldsymbol{\tau} \ldots)), \mathbf{X}_{\mathit{indices}}]\!] = ([\mathbf{x} \mapsto (x/\tau\text{->}x[\![\mathbf{x}/\boldsymbol{\tau}]\!] \ldots)])$

The specification for *dependencies* depends on the *indices* (§ 4) auxiliary meta-function to extract the relevant information from the scheme, and delegates to *dependencies**. The definition of *dependencies** traverses the cryptographic scheme, accumulating the dependencies for each definition. It finds the dependencies within expressions using another auxiliary meta-function *dependencies/e*, which we introduce next.

The following is the dependency graph for BLS:[38]

[38] See § B for a visualization of this graph.

```
((M ↦ ())
 (g ↦ ())
 (x ↦ ())
 (pk ↦ (g x))
 (sk ↦ (x))
 (σ ↦ (M sk))
 (h ↦ (M))
 (valid? ↦ (h pk σ g)))
```

And the following is the dependency graph for CL:

```
((M ↦ ())
 (g ↦ ())
 (x ↦ ())
 (y ↦ ())
 (X ↦ (g x))
 (Y ↦ (g y))
 (pk ↦ (X Y))
 (sk ↦ (x y))
 (a ↦ ())
 (b ↦ (a y))
 (c ↦ (a x M y))
 (σ ↦ (a b c))
 (valid? ↦ (Y a g b X M c)))
```

THE *dependencies/e* meta-function finds dependencies within an expression. It works by traversing the expression looking for variables that are not indices:[39]

$$
\begin{aligned}
&\textit{dependencies/e} : \_\ \mathbf{X}_{\text{indices}} \to \mathbf{X} \\
&\textit{dependencies/e}[\![\mathbf{x}, \mathbf{X}_{\text{indices}}]\!] \quad = () \\
&\quad \text{where } \in[\![\mathbf{x}, \mathbf{X}_{\text{indices}}]\!] \\
&\textit{dependencies/e}[\![\mathbf{x}, \mathbf{X}_{\text{indices}}]\!] \quad = (\mathbf{x}) \\
&\textit{dependencies/e}[\![(\mathbf{any}\ \ldots), \mathbf{X}_{\text{indices}}]\!] \quad = \cup[\![\textit{dependencies/e}[\![\mathbf{any}, \mathbf{X}_{\text{indices}}]\!], \ldots]\!] \\
&\textit{dependencies/e}[\![\mathbf{any}, \mathbf{X}_{\text{indices}}]\!] \quad = ()
\end{aligned}
$$

[39] Indices are introduced by expressions using **opl** operators, and are not dependencies.

USING THE data-flow graph determined by the meta-functions above, we can define relations which predicate on wether any variable in a set of potential dependencies is in fact a dependency for a given expression. This presents a more convenient method to query the graph and deals with details related to composite bindings. The following is the definition of the *depends?/any* relation:

$$
\frac{\begin{aligned}
&\textit{depends?/any*}[\![\textit{dependencies/e}[\![\mathbf{e}, \mathbf{X}_{\text{indices}}]\!], \\
&\qquad \cup[\![\mathbf{X}_{\text{potential-dependencies}}, \\
&\qquad\quad \textit{ranges*}[\![\textit{filter}[\![\mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad\qquad\qquad \mathbf{X}_{\text{potential-dependencies}}]]]\!], \\
&\qquad \mathbf{g}_{\text{dependencies/transitive-closure}}]\!]
\end{aligned}}{\begin{aligned}
&\textit{depends?/any}[\![\mathbf{e}, \mathbf{X}_{\text{potential-dependencies}}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}]\!]
\end{aligned}}
$$

$$\frac{\textit{overlap?}[\![\textit{ranges*}[\![\textit{filter}[\![\mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{X}_{\text{variable-references}}]\!]], \mathbf{X}_{\text{potential-dependencies}}]\!]}{\textit{depends?/any*}[\![\mathbf{X}_{\text{variable-references}}, \mathbf{X}_{\text{potential-dependencies}}, \mathbf{g}_{\text{dependencies/transitive-closure}}]\!]}$$

The *depends?/any* relation uses the *dependencies/e* meta-function defined above to determine the dependencies in the given expression and the *filter* and *ranges** auxiliary meta-functions (§ A) to extract the relevant information from composite bindings. A transitive closure of the graph of composite bindings is necessary because a dependence may refer to a component of a binding. For example, if pk is the composition of x and y, then depending on x entails (indirectly) depending on pk. Finally, the auxiliary *depends?/any** relation holds when there is an overlap between the set of potential dependencies in the query and the actual dependencies found in the expression.

A NATURAL COUNTERPART to the *depends?/any* relation defined above is *depends?/all*, which holds if *all* the potential dependencies in the query are actual dependencies:

$$\frac{\textit{depends?/any}[\![\mathbf{e}, (\mathbf{x}_{\text{potential-dependencies}}), \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}]\!] \quad \ldots}{\textit{depends?/all}[\![\mathbf{e}, (\mathbf{x}_{\text{potential-dependencies}}\ \ldots), \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}]\!]}$$

9 *Signature Scheme Verification*

ALL DEFINITIONS THUS FAR have been agnostic to the kind of cryptographic scheme, but, for one of our case studies (AutoBatch Generalization, § 12), we are particularly interested in signature schemes. We introduce in this section a verifier that checks whether a cryptographic scheme defined by the cryptographer is a signature scheme. This is an example of the sophisticated reasoning allowed by the analyses we described in the previous sections. We expect to extend SDL with verifiers for other kinds of cryptographic schemes in the future (§ 14).

A signature scheme is characterized by its algorithms, the data it computes, and by how data flows:

- The cryptographic scheme must define at least the algorithms keygen, sign and verify.
- keygen has no required inputs.
- keygen must output at least pk and sk.
- sign must receive as inputs at least sk and M.
- sign must output at least σ.
- The computation of σ must depend on sk and M.
- verify must receive as inputs at least pk, M and σ.

keygen	Generate Keys
pk	Public Key
sk	Secret (Private) Key
M	Message
σ	Signature

- verify must not receive as input sk.
- verify must output only valid?.
- valid? must be of type B (boolean).
- The computation of valid? must depend on pk, M and σ.
- All local variables (define-local and define-local/precomputed) must be used.
- The expression to compute valid? must have a specific form[40] (see below).

[40] The form expected by AutoBatch

THE FOLLOWING language extension specifies the format for expressions defining valid? in signature schemes:

vex ::= **veq** | (and **veq** ...)
veq ::= (= **ves ves**)
ves ::= **vef** | (× **vef** ...)
vef ::= **vet** | (↑ **vet e**) | (∏ **x e e vef**)
vet ::= **x** | (E **e e**)

THE *signature-scheme?* relation holds if the cryptographic scheme s is a signature scheme:

*signature-scheme?**⟦*definitions/expressions*⟦**s**⟧, *transitive-closure*⟦*definitions/composites*⟦**s**⟧⟧, *inputs/s*⟦**s**⟧, *locals/s*⟦**s**⟧, *outputs/s*⟦**s**⟧, *algorithms*⟦**s**⟧, *indices*⟦**s**⟧, *infer*⟦**s**⟧, *transitive-closure*⟦*dependencies*⟦**s**⟧⟧⟧

---

signature-scheme?⟦**s**⟧
∧⟦ ⊆ ⟦(keygen sign verify), $\mathbf{X}_{\text{algorithms}}$⟧,

⊆ */g*⟦([keygen ↦ ()]), $\mathbf{g}_{\text{inputs}}$⟧,
⊆ */g*⟦([keygen ↦ (pk sk)]), $\mathbf{g}_{\text{outputs}}$⟧,

⊆ */g*⟦([sign ↦ (sk M)]), $\mathbf{g}_{\text{inputs}}$⟧,
⊆ */g*⟦([sign ↦ (σ)]), $\mathbf{g}_{\text{outputs}}$⟧,
depends?/all⟦σ, (sk M), $\mathbf{g}_{\text{definitions/composites/transitive-closure}}$, $\mathbf{X}_{\text{indices}}$, $\mathbf{g}_{\text{dependencies/transitive-closure}}$⟧,

⊆ */g*⟦([verify ↦ (pk M σ)]), $\mathbf{g}_{\text{inputs}}$⟧,
¬⟦ ⊆ */g*⟦([verify ↦ (sk)]), $\mathbf{g}_{\text{inputs}}$⟧⟧,
= *?*⟦(valid?), *lookup*⟦verify, $\mathbf{g}_{\text{outputs}}$⟧⟧,
∈⟦*lookup*⟦valid?, **Γ**⟧, (B)⟧,
depends?/all⟦valid?, (pk M σ), $\mathbf{g}_{\text{definitions/composites/transitive-closure}}$, $\mathbf{X}_{\text{indices}}$, $\mathbf{g}_{\text{dependencies/transitive-closure}}$⟧,

⊆ ⟦*ranges**⟦$\mathbf{g}_{\text{locals}}$⟧, *ranges**⟦$\mathbf{g}_{\text{dependencies/transitive-closure}}$⟧⟧⟧ **vex** = *lookup*⟦valid?, $\mathbf{any}_{\text{definitions/expressions}}$⟧

---

*signature-scheme?**⟦$\mathbf{any}_{\text{definitions/expressions}}$, $\mathbf{g}_{\text{definitions/composites/transitive-closure}}$, $\mathbf{g}_{\text{inputs}}$, $\mathbf{g}_{\text{locals}}$, $\mathbf{g}_{\text{outputs}}$, $\mathbf{X}_{\text{algorithms}}$, $\mathbf{X}_{\text{indices}}$, **Γ**, $\mathbf{g}_{\text{dependencies/transitive-closure}}$⟧

The *signature-scheme?* relation depends on auxiliary meta-functions to extract relevant information from the cryptographic scheme s. And each precondition in *signature-scheme?** corresponds to one of the conditions mentioned above. Both SDL and CL as specified in § 3 are signature schemes according to these criteria.

10 Term Rewriting

ONE OF THE MAIN objectives in designing SDL is to support machine-aided design of cryptographic systems, and an important task in this area is to systematically rewrite terms in the language. In this section we introduce general techniques based on term rewriting systems,[41] and on the next sections (§ 11 and § 12) we present examples of these techniques in use.

[41] Klop 1992.

THE FIRST AND MOST USEFUL term rewriting technique is the meta-function, and, in Redex, there is convenient notation to define meta-functions. Consider the (arguably contrived) use case of transforming all additions into multiplication, an initial attempt would be the following:

```
(define-metafunction sdl
  [(+->× (+ any ...)) (× any ...)])
```

To use this meta-function, it is called like a function in the language:

```
> (term (+->× (+ 2 3)))
'(× 2 3)
```

We might want, however, for the +->× meta-function to ignore other kinds of inputs:

```
(define-metafunction sdl
  [(+->× (+ any ...)) (× any ...)]
  [(+->× any) any])
```

The definition above works because clauses in the meta-function definition are tested in order, and the first that matches is the meta-function result, so the **any** in the second clause only matches non-+ forms, for example:

```
> (term (+->× (/ 2 3)))
'(/ 2 3)
```

The final hurdle is that the operation of interest (addition, in our running example) might occur nested in other expressions and definitions, so we introduce a recursive descent traversal on the abstract syntax:

```
(define-metafunction sdl
  [(+->× (+ any ...)) (× any ...)]
  [(+->× (any ...)) ((+->× any) ...)]
  [(+->× any) any])
```

Note the second clause, which deconstructs a form, calls the meta-function `+->×` recursively on the parts and constructs a new form with the results. This addresses cases such as the following.

```
> (term (+->× (/ (+ 2 3) 3)))
'(/ (× 2 3) 3)
```

THERE ARE MORE sophisticated term rewritings for which meta-functions are not sufficient, for example, those requiring knowledge of the context or interactions with external tools like SMT solvers. For those cases, Redex provides an arbitrary extension mechanism called unquoting, denoted by the comma (,), which allows a meta-function to escape to arbitrary Racket code, including side-effecting computations.[42] In this context, terms in SDL are treated as S-expressions accessible using the term form. Consider the following example of a `+->×` variation which logs the addition operands while rewriting the terms:

[42] We avoid reaching for unquoting unless strictly required, to keep the specification self-contained.

```
(define-metafunction sdl
  [(+->× (+ any ...))
   ,(begin
      (displayln (term (any ...)))
      (term (× any ...)))]
  [(+->× (any ...)) ((+->× any) ...)]
  [(+->× any) any])
```

The following is an use example:

```
> (term (+->× (+ 2 3)))
(2 3)
'(× 2 3)
```

Besides the rewritten term—`'(× 2 3)`—the output also includes the printed operands: `(2 3)`. There is an important caveat to using this technique: Redex caches the meta-function applications to improve performance, and, depending on the use case, it might be best to turn caching off:

```
> (term (+->× (/ (+ 2 3) (+ 2 3))))
(2 3)
'(/ (× 2 3) (× 2 3))
```

```
> (parameterize ([caching-enabled? #f])
    (term (+->× (/ (+ 2 3) (+ 2 3)))))
(2 3)
(2 3)
'(/ (× 2 3) (× 2 3))
```

On the first example, the operands are only printed once, despite the rewriting happening twice, due to caching; whereas on the second example caching is disabled, so the operands are printed twice.

11 Case Study: Consistent Use of Group Operators

THERE ARE TWO notations for group operations in pairing-based cryptography: additive and multiplicative. To simplify SDL's ecosystem we introduce a normalization pass which converts the former into the latter, to be run between type inference and type checking (§ 6),[43] as an example of a simple transformation pass defined using our term rewriting techniques (§ 10).

[43] The type inferencer works over expressions both in additive and multiplicative notations, to inform the pass we introduce in this section, but the type checker expects multiplicative notation.

While both additive and multiplicative notations are valid, it is not possible to mingle them in the same cryptographic scheme, so we start by defining a relation *consistent-group-operators?* which only holds when the cryptographic scheme makes consistent use of group operators:

$$\frac{\textit{consistent-group-operators?}^{*}[\![\textit{additions} + \textit{subtractions}[\![\mathbf{s}]\!], \textit{multiplications} + \textit{divisions}[\![\mathbf{s}]\!], \textit{infer}[\![\mathbf{s}]\!]]\!]}{\dfrac{\textit{consistent-group-operators?}[\![\mathbf{s}]\!] \quad \neg\textit{op-uses-G/τ?}[\![\mathbf{e}_{\text{additions+subtractions}}, \Gamma]\!] \quad \ldots}{\textit{consistent-group-operators?}^{*}[\![(\mathbf{e}_{\text{additions+subtractions}} \ldots), (\mathbf{e}_{\text{multiplications+divisions}} \ldots), \Gamma]\!]} \quad \dfrac{\neg\textit{op-uses-G/τ?}[\![\mathbf{e}_{\text{multiplications+divisions}}, \Gamma]\!] \quad \ldots}{\textit{consistent-group-operators?}^{*}[\![(\mathbf{e}_{\text{additions+subtractions}} \ldots), (\mathbf{e}_{\text{multiplications+divisions}} \ldots), \Gamma]\!]}}$$

The top-level *consistent-group-operators?* relation relies on auxiliary meta-functions to extract the relevant expressions from the cryptographic scheme (the arithmetic operations), as well as on the type inferencer to recover the types of the operands. Finally, it depends on the *consistent-group-operators?** auxiliary relation, which only holds if either additive or multiplicative notation is unused, or, in other words, if it is not the case that both notations are used. The *consistent-group-operators?** auxiliary relation depends on the following relations to determine whether the operands are group elements:

$$\frac{\mathbf{G}/\boldsymbol{\tau} = \mathit{infer}^*/e[\![\mathbf{e}_1, \boldsymbol{\Gamma}]\!]}{\mathit{op\text{-}uses\text{-}G/\tau?}[\![(\mathbf{opn}\ \mathbf{e}_1\ \mathbf{e}_2\ \ldots), \boldsymbol{\Gamma}]\!]}$$

$$\frac{\mathbf{G}/\boldsymbol{\tau} = \mathit{infer}^*/e[\![\mathbf{e}_3, \boldsymbol{\Gamma}]\!]}{\mathit{op\text{-}uses\text{-}G/\tau?}[\![(\mathbf{opl}\ \mathbf{x}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3), \boldsymbol{\Gamma}]\!]} \qquad \frac{\neg[\![\mathit{op\text{-}uses\text{-}G/\tau?}[\![\mathbf{any}, \boldsymbol{\Gamma}]\!]]}{\neg \mathit{op\text{-}uses\text{-}G/\tau?}[\![\mathbf{any}, \boldsymbol{\Gamma}]\!]}$$

Next, we want to ascertain whether the scheme uses additive notation and needs to go through the normalization transformation, or if it can skip it:

$$\frac{\mathit{op\text{-}uses\text{-}G/\tau?}[\![\mathbf{e}_2, \mathit{infer}[\![\mathbf{s}]\!]]\!] \qquad (\mathbf{e}_1 \ldots\ \mathbf{e}_2\ \mathbf{e}_3\ \ldots) = \mathit{additions} + \mathit{subtractions}[\![\mathbf{s}]\!]}{\mathit{uses\text{-}+/\text{-}\text{-}group\text{-}operators?}[\![\mathbf{s}]\!]}$$

Finally, the *normalize-group-operators* meta-function uses term rewriting techniques to convert group operators using additive notation into multiplicative notation, which affects the forms +, - and ∑:

$\mathit{normalize\text{-}group\text{-}operators} : \mathbf{s} \rightarrow \mathbf{s}$

$\mathit{normalize\text{-}group\text{-}operators}[\![\mathbf{s}]\!] = \mathit{normalize\text{-}group\text{-}operators}^*[\![\mathbf{s}, \mathit{infer}[\![\mathbf{s}]\!]]\!]$

$\mathit{normalize\text{-}group\text{-}operators}^* : \_\ \boldsymbol{\Gamma} \rightarrow \_$

$\mathit{normalize\text{-}group\text{-}operators}^*[\![\mathbf{e}_1, \boldsymbol{\Gamma}]\!] = (\times\ \mathit{normalize\text{-}group\text{-}operators}^*[\![\mathbf{e}_2, \boldsymbol{\Gamma}]\!]\ \ldots)$
where $\mathit{op\text{-}uses\text{-}G/\tau?}[\![\mathbf{e}_1, \boldsymbol{\Gamma}]\!], \mathbf{e}_1 = (+\ \mathbf{e}_2\ \ldots)$

$\mathit{normalize\text{-}group\text{-}operators}^*[\![\mathbf{e}_1, \boldsymbol{\Gamma}]\!] = (\div\ \mathit{normalize\text{-}group\text{-}operators}^*[\![\mathbf{e}_2, \boldsymbol{\Gamma}]\!]\ \ldots)$
where $\mathit{op\text{-}uses\text{-}G/\tau?}[\![\mathbf{e}_1, \boldsymbol{\Gamma}]\!], \mathbf{e}_1 = (-\ \mathbf{e}_2\ \ldots)$

$\mathit{normalize\text{-}group\text{-}operators}^*[\![\mathbf{e}_1, \boldsymbol{\Gamma}]\!] = (\prod\ \mathit{normalize\text{-}group\text{-}operators}^*[\![\mathbf{e}_2, \boldsymbol{\Gamma}]\!]\ \ldots)$
where $\mathit{op\text{-}uses\text{-}G/\tau?}[\![\mathbf{e}_1, \boldsymbol{\Gamma}]\!], \mathbf{e}_1 = (\sum\ \mathbf{e}_2\ \ldots)$

$\mathit{normalize\text{-}group\text{-}operators}^*[\![(\mathbf{any}\ \ldots), \boldsymbol{\Gamma}]\!] = (\mathit{normalize\text{-}group\text{-}operators}^*[\![\mathbf{any}, \boldsymbol{\Gamma}]\!]\ \ldots)$

$\mathit{normalize\text{-}group\text{-}operators}^*[\![\mathbf{any}, \boldsymbol{\Gamma}]\!] = \mathbf{any}$

The following is an example of *normalize-group-operators* applied to the artificially oversimplified cryptographic scheme [(define-algorithm (keygen) (define-output pk (+ (random G1) (random G1))))]:

```
((define-algorithm (keygen) (define-output pk (× (random G1) (random G1)))))
```

And the following is an example of *normalize-group-operators* applied to the artificially oversimplified cryptographic scheme [(define-algorithm (keygen) (define-output pk (∑ x 0 1 (random G1))))]:

```
((define-algorithm (keygen) (define-output pk (∏ x 0 1 (random G1)))))
```

12 Case Study: AutoBatch Generalization

AUTOBATCH IS A tool to design batch signature verification schemes from regular signature verification schemes which helps cryptog-

raphers construct systems optimized for particular loads (batch sizes). AutoBatch's core is a collection of semantically-preserving term rewrite rules for verification equations that define a space of equivalent equations. We generalize the original implementation[44] in the context of our new SDL ecosystem.

[44] Akinyele et al. 2014a,b.

AUTOBATCH REQUIRES additional parameters to operate, besides the signature verification scheme and the verification equation contained in it. These parameters are provided in the language, using the **p** non-terminal, for example, for BLS:

```
((batch-size ↦ 100)
 (single-signer? ↦ #t)
 (security-parameter ↦ 80)
 (extra-untrusted-variables ↦ ()))
```

THE FIRST STEP in AutoBatch is to add small exponents δ to the untrusted terms in the verification equation $^{(M\,\sigma)}$. The small exponents are a list, and there is a Racket function `next-small-exponent` to generate indices in this list, which returns a new fresh index each time it is called, and we use Racket's parameters to define it.[45,46] The top level `add-small-exponents` function is a Racket function which initializes the current index counter, disables caching[47] and calls the auxiliary metafunction *add-small-exponents**:

[45] URL `https://docs.racket-lang.org/reference/parameters.html`

[46] The implementation is not included in this document, it consists of a function that increments the current small exponent index parameter and returns it.

[47] See § 10.

$$\textit{add-small-exponents*} : \mathbf{vex}\ \mathbf{g}_{\text{definitions/composites/transitive-closure}}\ \mathbf{X}_{\text{indices}}\ \mathbf{g}_{\text{dependencies/transitive-closure}}\ \mathbf{p} \to \mathbf{vex}$$

$$\begin{aligned}
&\textit{add-small-exponents*}[\![\mathbf{veq}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{p}]\!] \\
&\quad = \textit{add-small-exponent}[\![\mathbf{veq}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \texttt{(next-small-exponent)}, \mathbf{p}]\!] \\
&\text{where } \textit{needs-small-exponent?}[\![\mathbf{veq}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{p}]\!]
\end{aligned}$$

$$\begin{aligned}
&\textit{add-small-exponents*}[\![\mathbf{veq}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{p}]\!] \\
&\quad = \mathbf{veq}
\end{aligned}$$

$$\begin{aligned}
&\textit{add-small-exponents*}[\![(\text{and } \mathbf{veq} \ldots), \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{p}]\!] \\
&\quad = (\text{and } \textit{add-small-exponents*}[\![\mathbf{veq}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{p}]\!] \ldots)
\end{aligned}$$

$$\textit{add-small-exponent} : \mathbf{e}\ \mathbf{g}_{\text{definitions/composites/transitive-closure}}\ \mathbf{X}_{\text{indices}}\ \mathbf{g}_{\text{dependencies/transitive-closure}}\ \mathbf{e}_{\delta}\ \mathbf{p} \to \mathbf{e}$$

$$\begin{aligned}
&\textit{add-small-exponent}[\![(= \mathbf{ves} \ldots), \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{e}_{\delta}, \mathbf{p}]\!] \\
&\quad = (= \textit{add-small-exponent}[\![\mathbf{ves}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{e}_{\delta}, \mathbf{p}]\!] \ldots)
\end{aligned}$$

$$\begin{aligned}
&\textit{add-small-exponent}[\![(\times \mathbf{vef} \ldots), \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{e}_{\delta}, \mathbf{p}]\!] \\
&\quad = (\times \textit{add-small-exponent}[\![\mathbf{vef}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{e}_{\delta}, \mathbf{p}]\!] \ldots)
\end{aligned}$$

$$\begin{aligned}
&\textit{add-small-exponent}[\![\mathbf{vef}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{e}_{\delta}, \mathbf{p}]\!] \\
&\quad = (\uparrow \mathbf{vet}\ \textit{merge-}\times [\![\mathbf{e}_{\delta}, \mathbf{e}]\!]) \\
&\text{where } \textit{needs-small-exponent?}[\![\mathbf{vef}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \mathbf{vef} = (\uparrow \mathbf{vet}\ \mathbf{e}) \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{p}]\!]
\end{aligned}$$

$$\begin{aligned}
&\textit{add-small-exponent}[\![\mathbf{vet}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{e}_{\delta}, \mathbf{p}]\!] \\
&\quad = (\uparrow \mathbf{vet}\ \mathbf{e}_{\delta}) \\
&\text{where } \textit{needs-small-exponent?}[\![\mathbf{vet}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{p}]\!]
\end{aligned}$$

$$\begin{aligned}
&\textit{add-small-exponent}[\![\mathbf{any}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\
&\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{e}_{\delta}, \mathbf{p}]\!] \\
&\quad = \mathbf{any}
\end{aligned}$$

The definitions of *add-small-exponents** and its auxiliary meta-function *add-small-exponent* follow the term rewriting techniques from § 10, and depend on the *needs-small-exponent?* relation introduced next. The form with pink background represents unquoting (escaping to arbitrary Racket code). The following is an example of `add-small-exponents` applied to (= (E h pk) (E σ g)):

```
(= (↑ (E h pk) (@ δ 1)) (↑ (E σ g) (@ δ 1)))
```

THE `add-small-exponents` function defined above depends on the *needs-small-exponent?* which holds if and only if the given argument is untrusted:

$$\frac{\begin{aligned}&\textit{depends?/any}[\![\mathbf{e}, \cup[\![\mathbf{X}_{\text{untrusted}}, \textit{lookup*}[\![\text{extra-untrusted-variables}, \mathbf{p}, ()]\!]]\!], \\ &\qquad \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\ &\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}]\!]\end{aligned}}{\begin{aligned}&\textit{needs-small-exponent?}[\![\mathbf{e}, \mathbf{g}_{\text{definitions/composites/transitive-closure}}, \\ &\qquad \mathbf{X}_{\text{indices}}, \mathbf{g}_{\text{dependencies/transitive-closure}}, \mathbf{p}]\!]\end{aligned}}$$

AFTER HAVING ADDED the small exponents, AutoBatch has to consolidate the equations that are part of the signature verification, which amounts to combining the equations sides by multiplying

them:

consolidate-equation : **vex** → **veq**

consolidate-equation⟦(and **veq**)⟧ = **veq**

consolidate-equation⟦(and (= $\mathbf{ves}_{1\text{-}l}$ $\mathbf{ves}_{1\text{-}r}$) (= $\mathbf{ves}_{2\text{-}l}$ $\mathbf{ves}_{2\text{-}r}$) **veq** ...)⟧ = *consolidate-equation*⟦(and (= *merge-*×⟦$\mathbf{ves}_{1\text{-}l}$, *invert*⟦$\mathbf{ves}_{1\text{-}r}$⟧⟧ *merge-*×⟦$\mathbf{ves}_{2\text{-}l}$, *invert*⟦$\mathbf{ves}_{2\text{-}r}$⟧⟧) **veq** ...)⟧

consolidate-equation⟦**veq**⟧ = **veq**

The *consolidate-equation* meta-function depends on a few auxiliary arithmetic meta-functions (§ A). The following is an example of *consolidate-equation* applied to (and (= (↑ (E a b) (@ δ 1)) (↑ (E c d) (@ δ 1))) (= (↑ (E e f) (@ δ 2)) (↑ (E g h) (@ δ 2)))):

```
(=
 (× (↑ (E a b) (@ δ 1)) (↑ (E c d) (- (@ δ 1))))
 (× (↑ (E e f) (@ δ 2)) (↑ (E g h) (- (@ δ 2)))))
```

THE NEXT STEP on the AutoBatch pipeline is to convert a signature scheme that verifies a single signature into a batch verifier. It is necessary to add indices to the terms that vary, for example, messages and public keys (unless there is a parameter saying there is only a single signer):

batch : **veq** $\mathbf{g}_{\text{dependencies/transitive-closure}}$ **p** → **veq**

batch⟦(= $\mathbf{ves}_1$ $\mathbf{ves}_2$), $\mathbf{g}_{\text{dependencies/transitive-closure}}$, **p**⟧ = (= (∏ z 1 η *add-indices*⟦$\mathbf{ves}_1$, $\mathbf{g}_{\text{dependencies/transitive-closure}}$, $\mathbf{X}_{\text{indexable}}$⟧) (∏ z 1 η *add-indices*⟦$\mathbf{ves}_2$, $\mathbf{g}_{\text{dependencies/transitive-closure}}$, $\mathbf{X}_{\text{indexable}}$⟧))

where $\mathbf{X}_{\text{indexable}}$ = *if*⟦*lookup*⟦single-signer?, **p**⟧, (M σ), (M σ pk)⟧

add-indices : _ $\mathbf{g}_{\text{dependencies/transitive-closure}}$ $\mathbf{X}_{\text{indexable}}$ → _

add-indices⟦(@ δ **integer**), $\mathbf{g}_{\text{dependencies/transitive-closure}}$, $\mathbf{X}_{\text{indexable}}$⟧ = (@ (@ δ **integer**) z)

add-indices⟦**x**, $\mathbf{g}_{\text{dependencies/transitive-closure}}$, $\mathbf{X}_{\text{indexable}}$⟧ = (@ $\mathbf{x}_{\text{list}}$ z)

where *overlap?*⟦*lookup*⟦**x**, $\mathbf{g}_{\text{dependencies/transitive-closure}}$⟧, $\mathbf{X}_{\text{indexable}}$⟧, $\mathbf{x}_{\text{list}}$ = `(string->symbol (~a x "-list"))`

add-indices⟦(**any** ...), $\mathbf{g}_{\text{dependencies/transitive-closure}}$, $\mathbf{X}_{\text{indexable}}$⟧ = (*add-indices*⟦**any**, $\mathbf{g}_{\text{dependencies/transitive-closure}}$, $\mathbf{X}_{\text{indexable}}$⟧ ...)

add-indices⟦**any**, $\mathbf{g}_{\text{dependencies/transitive-closure}}$, $\mathbf{X}_{\text{indexable}}$⟧ = **any**

The following is an example of the *batch* meta-function applied to the equation (= (↑ (E h pk) (@ δ 1)) (↑ (E σ g) (@ δ 1))):

```
(=
 (∏ z 1 η (↑ (E (@ h-list z) pk) (@ (@ δ 1) z)))
 (∏ z 1 η (↑ (E (@ σ-list z) g) (@ (@ δ 1) z))))
```

AT THIS POINT the scheme is suited for verification of a batch of signatures, but it has to be optimized for the particular load in terms of batch size. AutoBatch's approach is to repeatedly apply a collection of semantic-preserving transformations to the equation, looking for the optimal version according to some objective function which, for example, assigns greater cost to generating group elements than it does to adding two numbers together. The rewrite rules follow the rules of arithmetic and properties of bilinear maps, for example,

moving an exponent inside a pairing: $e(a,b)^c \rightarrow e(a^c,b)$. To facilitate writing these rules, we introduce a new form `define-technique`,[48] for example:

```
(define-technique (move-exponent-inside-pairing)
  [((↑ (E e_1 e_2) e_3)) (E (↑ e_1 e_3) e_2)])
```

The following is an example of the rule applied to the term (= (∏ z 1 η (↑ (E (@ h-list z) pk) (@ (@ δ 1) z))) (∏ z 1 η (↑ (E (@ σ-list z) g) (@ (@ δ 1) z)))):

```
(=
 (∏ z 1 η (E (↑ (@ h-list z) (@ (@ δ 1) z)) pk))
 (∏ z 1 η (E (↑ (@ σ-list z) (@ (@ δ 1) z)) g)))
```

In the previous AutoBatch implementation, rules of this kind required hundreds of lines of code, and in our new SDL framework it can be expressed in two. But the work on this application is not finished (§ 14), in particular, the driver of the goal-directed search and the heuristics to guide it are left as future work, as the scope of this report is the foundational work in SDL.

[48] The definition of `define-technique` is not in this document, but is available in the implementation.

13 Cryptographic Primitives Interface

CRYPTOGRAPHIC SCHEMES defined in SDL can be converted into executable code in the form of libraries for other programming languages, for example, Rust and Python. The same term rewriting techniques introduced above (§ 10) applies: these code-generation back-ends traverse the cryptographic scheme and generate code in the target language. The resulting code depends on auxiliary libraries to provide the SDL cryptographic primitives: bilinear maps (`E`), types for group elements (`G1`, `G2` and `GT`), and so forth.

The development of back-ends is beyond the scope of this qualifying project (§ 15), which focuses on the core SDL design. But we introduce the specification of an interface between the generated code and the auxiliary libraries. The code-generation back-ends must output code using this interface, and auxiliary libraries must implement it. This specification is high-level, and instances should adapt it to the features provided by the target language. In particular, implementers should consider the following aspects of the interface, which some languages do not support:

- Identifiers including non-ASCII Unicode code points.
- Overloaded operators.
- Parametric polymorphic list types.

WE ASSUME that the target language includes types for strings, integers, booleans and lists. In addition, the auxiliary library must

provide types for:

- ZR: Elements of the ring of integers modulo r.
- G1, G2 and GT: Group elements.

WE ASSUME that the target language includes the basic operations over strings, integers, booleans and lists. In addition, the auxiliary library must provide operations for:[49]

[49] Operation : Type: Description

- G1.hash : → G1: Hashing over elements of G1.
- G2.hash : → G2: Hashing over elements of G2.
- String.random : → String: Draw a random string.
- Integer.random : → Integer: Draw a random integer.
- Boolean.random : → Boolean: Draw a random boolean.
- ZR.random : → ZR: Draw a random element of the ring of integers modulo r.
- G1.random : → G1: Draw a random element from group G1.
- G2.random : → G2: Draw a random element from group G2.
- GT.random : → GT: Draw a random element from group GT.
- G1.init : → G1: Return an initial element from group G1.
- G2.init : → G2: Return an initial element from group G2.
- GT.init : → GT: Return an initial element from group GT.
- ZR.init : → ZR: Return an initial element of the ring of integers modulo r.
- + : ZR ZR → ZR: Addition over elements of the ring of integers modulo r.
- - : ZR ZR → ZR: Subtraction over elements of the ring of integers modulo r.
- × : ZR ZR → ZR: Multiplication over elements of the ring of integers modulo r.
- × : G1 G1 → G1: Multiplication over elements of the group G1.
- × : G2 G2 → G2: Multiplication over elements of the group G2.
- × : GT GT → GT: Multiplication over elements of the group GT.
- ÷ : ZR ZR → ZR: Division over elements of the ring of integers modulo r.
- ÷ : G1 G1 → G1: Division over elements of the group G1.
- ÷ : G2 G2 → G2: Division over elements of the group G2.
- ÷ : GT GT → GT: Division over elements of the group GT.
- ↑ : I ZR → I: Exponentiation of integer to the power of element of the ring of integers modulo r.
- ↑ : ZR ZR → ZR: Exponentiation over elements of the ring of integers modulo r.
- ↑ : ZR I → ZR: Exponentiation of element of the ring of integers modulo r to the power of integer.

- ↑ : G1 I → G1: Exponentiation of element of group G1 to the power of integer.
- ↑ : G2 I → G2: Exponentiation of element of group G2 to the power of integer.
- ↑ : GT I → GT: Exponentiation of element of group GT to the power of integer.
- ↑ : G1 ZR → G1: Exponentiation of element of group G1 to the power of element of the ring of integers modulo r.
- ↑ : G2 ZR → G2: Exponentiation of element of group G2 to the power of element of the ring of integers modulo r.
- ↑ : GT ZR → GT: Exponentiation of element of group GT to the power of element of the ring of integers modulo r.
- · : Any … → String:[50] Bit-wise concatenation.
- e : G1 G1 → GT: Bilinear map (e) in symmetric setting (§ 7).
- e : G1 G2 → GT: Bilinear map (e) in asymmetric setting (§ 7).

[50] The ellipsis (…) means "zero or more repetitions of the previous form."

14 Future Work

THERE ARE SEVERAL dimensions in which we plan to proceed with our work. The first is developing a front-end for SDL consisting of a concrete syntax and accompanying tools. The goal of this concrete syntax is to be familiar to cryptographers, and the following is a preliminary sketch of a program defining BLS in it:

```
keygen() {
  g = random(G2)
  x = random(ZR)
  pk = g^x
  sk = x

  return pk, sk, g
}

sign(sk, M) {
  sig = H(M, G1)^sk
  return sig
}

verify(pk, M, sig, g) {
  h = H(M, G1)
  return (e(h, pk) =?= e(sig, g))
}
```

The plans for this surface language include extended features in the form of syntax sugar, for example, user-defined functions that

will be inlined by the front-end before converting to SDL, in a process similar to macro expansion.[51] We also anticipate the development of a lightweight module system similar to the one in Python.

Among the auxiliary tools, the most important is a command-line interface (CLI) that integrates the modules we described in this document and allows users to control them. Additionally, we consider bringing SDL into the Racket ecosystem[52] by creating a `#lang` for it. This would pave the way to build more tools which make SDL more ergonomic, including a syntax highlighter, an auto-indenter and inline error markers, which we would reuse from DrRacket.[53] Also, we plan on working on more informative error messages, and our course of action is instrumenting the executable specification we presented in this report with hidden side-conditions. Finally, we will produce user documentation which goes less in depth about the implementation than this document and is geared towards cryptographers; this documentation will take the form of a manual and a website.

ON THE COMPILER BACK-END, we will develop code generators that transform schemes in SDL into libraries for general-purpose programming languages, including Rust, JavaScript, Python, C++ and so forth. As a special case, we consider targeting LaTeX to assist cryptographers inspecting schemes and writing papers. The techniques to build these back-ends is term rewriting (§ 13), and the generated code will treat cryptographic primitives as black boxes by following an interface (§ 13). Additionally, we also contemplate developing a back-end in Racket through linguistic reuse.[54]

AFTER THIS FOUNDATIONAL WORK is complete, we will develop more applications, completing the missing aspects of the AutoBatch generalization (§ 12), for example, the heuristics to guide the search, and revisiting the work in AutoGroup+,[55] which will exercise SDL's capability of escaping to Racket as a general extensibility mechanism, because it depends on an SMT solver to find solutions to constraint satisfaction sub-problems. We foresee applications for our language in CloudSource[56] and zk-SNARKs in Zcash,[57] as well as in the study of cryptographic protocols, for example, TLS, which would be conducted by adding a SDL code-generation back-end for tools such as FlexTLS.[58] Finally, we expect to use SDL in the development of adaptive cryptographic systems which generate cryptographic schemes just-in-time, responding to demands in the environment; for example, the batch signature verification in car-to-car communications, in which the batch size varies according to the situation and the schemes could be optimized correspondingly.

[51] This limits the capabilities of these user-defined functions; for example, they cannot be unboundedly recursive.

[52] Felleisen et al. 2015.

[53] Findler et al. 2002.

[54] Krishnamurthi 2001.

[55] Akinyele et al. 2015.

[56] Joseph Ayo Akinyele et al. 2014, Green et al. 2011.

[57] Sasson et al. 2014.

[58] Beurdouche et al. 2015.

THERE STILL ROOM for improvement in SDL's core, which we plan to address in the future. This spans from minor features—for example, let function arguments have optional default values—to major ones—for example, a more sophisticated type system which incorporates notions of privacy and trust among its properties.

EXTENSIBILITY IS A CORE TENET IN SDL design, and there are aspects left for us to explore in the time to come. We plan on building auxiliary DSLs on top of Redex to: aid on the definitions of other kinds of cryptographic schemes, beyond signature schemes (§ 9); specify transformation rules and heuristics for AutoBatch; and code-generation back-ends. Along these lines, we also want to devise mechanisms for cryptographers to bring their own cryptographic primitives into the system, possibly apart from pairing-based encryption. These pluggable components would include information about the types of the operations, to inform the type system, and interface with the code-generation process.

FINALLY, in terms of theory, we want to pursue frameworks for specifying semantics, which would allow us to define SDL's semantics in a self-contained manner, instead of relying on translation to general-purpose programming languages. At this point, we will be in a position to explore meta-theoretic properties of the language and related definitions, for example, the soundness of its type system.

15 Related Work

THE WORK CLOSEST TO SDL is the auto-tools family of machine-aided cryptography design.[59] They introduced SDL's first version, of which the language in this report is a revision, addressing shortcomings including the absence of a formal specification as well as constructs such as deeply nested loops. Also along these lines there is Cryptol, a DSL for cryptographic systems[60]. The first noticeable difference between Cryptol and SDL is that Cryptol was designed for hardware circuits, whereas SDL attempts to be more general, despite the current implementation focusing on pairing-based encryption. We do consider drawing inspiration from Cryptol for the upcoming development of SDL's concrete syntax (§ 14). Finally, the techniques used to specify SDL and operations on it, including term rewriting (§ 10), come from Redex.[61]

[59] Akinyele et al. 2014a,b, 2013b, 2015, Akinyele 2013.

[60] Lewis.

[61] Felleisen et al. 2009.

16 Conclusion

WE INTRODUCED SDL, A DSL for cryptographic systems which resembles the mathematical specifications found in papers and serves as foundation for the machine-aided design of cryptographic schemes. The SDL specification includes a grammar for its abstract syntax, a collection of syntactic analyses, a well-formedness condition which verifies bindings and scoping rules, a type system, symmetry analysis, dependence analysis, a data-flow analysis to verify that a given scheme is a signature scheme, techniques for rewriting terms in the language, and an interface with cryptographic primitives to be provided in languages targeted by code generators. We also explored two case studies, the first being a simple example of term rewriting to guarantee the consistent use of multiplicative notation in group operators, and the second being a generalization of AutoBatch.

A core tenet in SDL design is extensibility. While the current version is focused on pairing-based signature schemes, we anticipate the language being able to support other kinds of cryptographic schemes, including public-key cryptography, identity-based encryption, attribute-based encryption, cryptographic protocols and so forth.

The specification found in this document is executable, for being based on Redex, and there is a suite of tests to assert that this reference implementation conforms to cryptographer's expectations. Besides the core language, it is left for future work the other components of a compiler for a fully-featured language, including a front-end for a concrete syntax (lexer, parser, and so forth), as well as back-ends for compiling schemes in SDL into libraries for general-purpose programming languages. The main issue to be addressed in the near future is to make the language more ergonomic, by providing cryptographers with a concrete syntax and editor support, including syntax highlighting and more informative error messages.

References

URL `https://docs.racket-lang.org/reference/parameters.html`.

URL `https://github.com/jakinyele/sdl-tools/tree/qualifying-project-report`.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2nd edition, 2007.

J. A. Akinyele, G. Barthe, B. Grégoire, B. Schmidt, and P. Y. Strub. Certified Synthesis of Efficient Batch Verifiers. In *2014 IEEE 27th*

Computer Security Foundations Symposium, pages 153–165, July 2014a.

Joseph Akinyele. *Enabling Machine-Aided Cryptographic Design*. Thesis, November 2013.

Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. Charm: A Framework for Rapidly Prototyping Cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, June 2013a.

Joseph A. Akinyele, Matthew Green, and Susan Hohenberger. Using SMT Solvers to Automate Design Tasks for Encryption and Signature Schemes. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 399–410. ACM, 2013b.

Joseph A. Akinyele, Matthew Green, Susan Hohenberger, and Matthew Pagano. Machine-Generated Algorithms, Proofs and Software for the Batch Verification of Digital Signature Schemes. *J. Comput. Secur.*, 22(6):867–912, November 2014b.

Joseph A. Akinyele, Christina Garman, and Susan Hohenberger. Automating Fast and Secure Translations from Type-I to Type-III Pairing Schemes. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1370–1381. ACM, 2015.

John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-Policy Attribute-Based Encryption. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 321–334. IEEE, 2007.

Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan. FLEXTLS: A Tool for Testing TLS Implementations. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT'15. USENIX Association, 2015.

Dan Boneh and Matthew K. Franklin. Identity-Based Encryption from the Weil Pairing. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 213–229. Springer-Verlag, 2001.

Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

Jan Camenisch and Anna Lysyanskaya. *Signature Schemes and Anonymous Credentials from Bilinear Maps*, pages 56–72. Springer Berlin Heidelberg, 2004.

John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—Open Source Graph Drawing Tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *SNAPL*, 2015.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: A programming environment for scheme. *J. Funct. Program.*, 12(2):159–182, March 2002.

Matthew Green, Susan Hohenberger, and Brent Waters. Outsourcing the Decryption of ABE Ciphertexts. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 34–34. USENIX Association, 2011.

Viet Tung Hoang, Jonathan Katz, and Alex J. Malozemoff. Automated Analysis and Synthesis of Authenticated Encryption Schemes. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 84–95. ACM, 2015.

Paul Hudak. Domain-Specific Languages. *Handbook of Programming Languages*, 3(39-60):21, 1997.

Joseph Ayo Akinyele, Matthew W. Pagano, Matthew Green, and Avi Rubin. Automatically Generating Outsourced Decryption for Pairing-Based Encryption Schemes. 2014.

Andrew W. Keep. *A Nanopass Framework for Commercial Compiler Development*. PhD thesis, Indiana University, 2013.

Jan Willem Klop. Term Rewriting Systems. *Handbook of logic in computer science*, 2:1–116, 1992.

Neal Koblitz and Alfred Menezes. Pairing-Based Cryptography at High Security Levels. *Lecture Notes in Computer Science*, 3796:13, 2005.

Shriram Krishnamurthi. *Linguistic Reuse*. PhD thesis, 2001.

Jeff Lewis. Cryptol: Specification, implementation and verification of high-grade cryptographic applications. In *Proceedings of the 2007 ACM Workshop on Formal Methods in Security Engineering*, FMSE '07, pages 41–41. ACM.

Alex J. Malozemoff, Jonathan Katz, and Matthew D. Green. Automated Analysis and Synthesis of Block-Cipher Modes of Operation. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, pages 140–152. IEEE Computer Society, 2014.

Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A Nanopass Infrastructure for Compiler Education. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 201–212. ACM, 2004.

Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society, 2014.

A Auxiliary Definitions

THE AUXILIARY data structures in the SDL specification are defined as language extensions, and operations on them are constructively defined using syntactic manipulations. Most operations are standard, mathematical relations which we represented as Redex meta-functions and relations. In this appendix we introduce these definitions, which were used throughout the rest of the document.

A.1 Lists

THE $all = ?$ relation holds if and only if all elements in the list are equal:

$$\frac{}{all = ?[\![()]\!]}$$

$$\frac{}{all = ?[\![(\mathbf{any})]\!]}$$

$$\frac{all = ?[\![(\mathbf{any}_1\ \mathbf{any}_2\ \ldots)]\!]}{all = ?[\![(\mathbf{any}_1\ \mathbf{any}_1\ \mathbf{any}_2\ \ldots)]\!]}$$

THE *unique* meta-function remove duplicate elements in a list:

$$unique : (\_ \ldots) \to (\_ \ldots)$$
$$unique[\![()]\!] = ()$$
$$unique[\![(\mathbf{any}_1 \ldots \mathbf{any}_2)]\!] = unique[\![(\mathbf{any}_1 \ldots)]\!] \text{ where } \in[\![\mathbf{any}_2, (\mathbf{any}_1 \ldots)]\!]$$
$$unique[\![(\mathbf{any}_1 \ldots \mathbf{any}_2)]\!] = \cup^*[\![unique[\![(\mathbf{any}_1 \ldots)]\!], (\mathbf{any}_2)]\!]$$

THE *unique?* relation holds if and only if the list does not contain duplicate elements:

$$unique?[\![(\mathbf{any}_{!\_} \ldots)]\!]$$

THE $\cup^*$ meta-function returns the concatenation (union with repeated elements) of the given lists:

$$\cup^* : (\_ \ldots) \ldots \to (\_ \ldots)$$
$$\cup^*[\![(\mathbf{any} \ldots), \ldots]\!] = (\mathbf{any} \ldots \ldots)$$

A.2 Sets

SETS are represented as lists without repeated elements.

THE *empty?* relation holds only for the empty set (or the empty list):

$$=?[\![\mathbf{any}, ()]\!]$$
$$empty?[\![\mathbf{any}]\!]$$

THE $\cup$ meta-function returns the union of the given sets:

$$\cup : (\_ \ldots) \ldots \to (\_ \ldots)$$
$$\cup[\![\mathbf{any}, \ldots]\!] = unique[\![\cup^*[\![\mathbf{any}, \ldots]]\!]$$

THE $\cap$ meta-function returns the intersection of the given sets (or lists):

$$\cap : (\_ \ldots) \ldots \to (\_ \ldots)$$
$$\cap[\![]\!] = ()$$
$$\cap[\![(), \mathbf{any}, \ldots]\!] = ()$$
$$\cap[\![(\mathbf{any}_1\ \mathbf{any}_2 \ldots), \mathbf{any}_3, \ldots]\!] = \cup[\![(\mathbf{any}_1), \cap[\![(\mathbf{any}_2 \ldots), \mathbf{any}_3, \ldots]]\!] \text{ where } (\#t \ldots) = (\in[\![\mathbf{any}_1, \mathbf{any}_3]\!] \ldots)$$
$$\cap[\![(\mathbf{any}_1\ \mathbf{any}_2 \ldots), \mathbf{any}_3, \ldots]\!] = \cap[\![(\mathbf{any}_2 \ldots), \mathbf{any}_3, \ldots]\!]$$

THE $\in$ relation holds if and only if the element is in the set (or list):

$$\in[\![\mathbf{any}_1, (\mathbf{any}_2 \ldots \mathbf{any}_1\ \mathbf{any}_3 \ldots)]\!]$$

THE $\notin$ relation holds if and only if the $\in$ relation does not hold:

$$\neg[\![\in[\![\mathbf{any}_1, \mathbf{any}_2]]\!]$$
$$\notin[\![\mathbf{any}_1, \mathbf{any}_2]\!]$$

THE $\subseteq$ relation holds if and only if the set on the left is a subset of (or equal to) the set on the right:

$$\frac{}{\subseteq [\![(), \mathbf{any}]\!]}$$

$$\frac{\in [\![\mathbf{any}_1, (\mathbf{any}_3 \ldots)]\!] \qquad \subseteq [\![(\mathbf{any}_2 \ldots), (\mathbf{any}_3 \ldots)]\!]}{\subseteq [\![(\mathbf{any}_1\ \mathbf{any}_2 \ldots), (\mathbf{any}_3 \ldots)]\!]}$$

THE $=?$ relation holds if and only if the sets are equal:

$$\frac{\subseteq [\![\mathbf{any}_1, \mathbf{any}_2]\!] \qquad \subseteq [\![\mathbf{any}_2, \mathbf{any}_1]\!]}{=?[\![\mathbf{any}_1, \mathbf{any}_2]\!]}$$

THE *set--* meta-function returns the set without the given element:

$$\textit{set--} : (\_ \ldots)\ (\_ \ldots) \rightarrow (\_ \ldots)$$

$$\textit{set--}[\![(), \mathbf{any}]\!] = ()$$

$$\textit{set--}[\![(\mathbf{any}_1\ \mathbf{any}_2 \ldots), \mathbf{any}_3]\!] = \textit{set--}[\![(\mathbf{any}_2 \ldots), \mathbf{any}_3]\!]$$

where $\in [\![\mathbf{any}_1, \mathbf{any}_3]\!]$

$$\textit{set--}[\![(\mathbf{any}_1\ \mathbf{any}_2 \ldots), \mathbf{any}_3]\!] = \cup [\![(\mathbf{any}_1), \textit{set--}[\![(\mathbf{any}_2 \ldots), \mathbf{any}_3]\!]]\!]$$

THE *disjunct?* relation holds if and only if the sets are disjunct (no elements in common):

$$\frac{\textit{empty?}[\![\cap [\![\mathbf{any}_1, \mathbf{any}_2]\!]]\!]}{\textit{disjunct?}[\![\mathbf{any}_1, \mathbf{any}_2]\!]}$$

THE *overlap?* relation holds if and only if there is at least one element in common in the sets:

$$\frac{\neg [\![\textit{disjunct?}[\![\mathbf{any}_1, \mathbf{any}_2]\!]]\!]}{\textit{overlap?}[\![\mathbf{any}_1, \mathbf{any}_2]\!]}$$

A.3 (Multi-)Maps

(MULTI-)MAPS are sets of mappings (pairs indexed by the left element). They can also be interpreted as enumerations of cases of partial functions.

THE *lookup* meta-function returns the element(s) at the given index:

$$\textit{lookup} : \mathbf{x}\ ([\mathbf{x} \mapsto \_] \ldots) \rightarrow \_$$

$$\textit{lookup}[\![\mathbf{x}, ([\mathbf{x}_1 \mapsto \mathbf{any}_1] \ldots [\mathbf{x} \mapsto \mathbf{any}]\ [\mathbf{x}_2 \mapsto \mathbf{any}_2] \ldots)]\!] = \mathbf{any}$$

THE *lookup** meta-function returns the element(s) at the given index if it occurs in the map, otherwise it returns the given default value:

$$lookup^* : \mathbf{x}\ ([\mathbf{x} \mapsto \_]\ ...)\ \_ \to \_$$

$$lookup^*[\![\mathbf{x}, ([\mathbf{x}_1 \mapsto \mathbf{any}_1]\ ...\ [\mathbf{x} \mapsto \mathbf{any}]\ [\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), \mathbf{any}_3]\!] = \mathbf{any}$$

$$lookup^*[\![\mathbf{x}, \mathbf{any}_1, \mathbf{any}_2]\!] = \mathbf{any}_2$$

THE *filter* meta-function filters the given indexes in the map:

$$filter : ([\mathbf{x} \mapsto \_]\ ...)\ (\mathbf{x}\ ...) \to ([\mathbf{x} \mapsto \_]\ ...)$$

$$filter[\![(), \mathbf{any}_3]\!] = ()$$

$$filter[\![([\mathbf{x}_1 \mapsto \mathbf{any}_1]\ [\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), \mathbf{any}_3]\!] = \cup[\![([\mathbf{x}_1 \mapsto \mathbf{any}_1]), filter[\![([\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), \mathbf{any}_3]\!]]\!]$$

where $\in[\![\mathbf{x}_1, \mathbf{any}_3]\!]$

$$filter[\![([\mathbf{x}_1 \mapsto \mathbf{any}_1]\ [\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), \mathbf{any}_3]\!] = filter[\![([\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), \mathbf{any}_3]\!]$$

THE *domain* meta-function returns the indexes in the map:

$$domain : ([\mathbf{x} \mapsto \_]\ ...) \to (\_\ ...)$$

$$domain[\![([\mathbf{x} \mapsto \mathbf{any}]\ ...)]\!] = (\mathbf{x}\ ...)$$

THE *ranges* meta-function returns the ranges in the map separately:

$$ranges : ([\mathbf{x} \mapsto \_]\ ...) \to (\_\ ...)$$

$$ranges[\![([\mathbf{x} \mapsto \mathbf{any}]\ ...)]\!] = (\mathbf{any}\ ...)$$

THE *ranges** meta-function returns an aggregation (union) of the ranges in the map:

$$ranges^* : ([\mathbf{x} \mapsto (\_\ ...)]\ ...) \to (\_\ ...)$$

$$ranges^*[\![([\mathbf{x} \mapsto \mathbf{any}]\ ...)]\!] = \cup[\![\mathbf{any}, ...]\!]$$

A.4 Graphs

GRAPHS are represented as multi-maps.

THE $\cup$ meta-function returns an union of the graphs (the point-wise union of the edges):

$$\cup/g : ([\mathbf{x} \mapsto (\_\ ...)]\ ...)\ ([\mathbf{x} \mapsto (\_\ ...)]\ ...) \to ([\mathbf{x} \mapsto (\_\ ...)]\ ...)$$

$$\cup/g[\![(), \mathbf{any}]\!] = \mathbf{any}$$

$$\cup/g[\![([\mathbf{x}_1 \mapsto \mathbf{any}_1]\ [\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), ([\mathbf{x}_3 \mapsto \mathbf{any}_3]\ ...\ [\mathbf{x}_1 \mapsto \mathbf{any}_4]\ [\mathbf{x}_5 \mapsto \mathbf{any}_5]\ ...)]\!] = \cup[\![([\mathbf{x}_1 \mapsto \cup[\![\mathbf{any}_1, \mathbf{any}_4]\!]]), \cup/g[\![([\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), ([\mathbf{x}_3 \mapsto \mathbf{any}_3]\ ...\ [\mathbf{x}_5 \mapsto \mathbf{any}_5]\ ...)]\!]]\!]$$

$$\cup/g[\![([\mathbf{x}_1 \mapsto \mathbf{any}_1]\ [\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), \mathbf{any}_3]\!] = \cup[\![([\mathbf{x}_1 \mapsto \mathbf{any}_1]), \cup/g[\![([\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), \mathbf{any}_3]\!]]\!]$$

THE $\subseteq/g$ relation holds if and only if the graph on the left is (point-wise) a subset of (or equal to) the graph on the right:

$$\frac{}{\subseteq/g[\![(), \mathbf{any}]\!]}$$

$$\frac{\subseteq[\![\mathbf{any}_1, \mathbf{any}_5]\!] \qquad \subseteq/g[\![([\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), \mathbf{any}_3]\!] \qquad \mathbf{any}_3 = ([\mathbf{x}_4 \mapsto \mathbf{any}_4]\ ...\ [\mathbf{x} \mapsto \mathbf{any}_5]\ [\mathbf{x}_6 \mapsto \mathbf{any}_6]\ ...)}{\subseteq/g[\![([\mathbf{x} \mapsto \mathbf{any}_1]\ [\mathbf{x}_2 \mapsto \mathbf{any}_2]\ ...), \mathbf{any}_3]\!]}$$

THE *edges* meta-function returns an enumeration of the edges in the graph:

edges : ([**x** ↦ (_ ...)] ...) → ([**x** ↦ _] ...)

edges⟦([**x** ↦ (**any** ...)])⟧ = ([**x** ↦ **any**] ...)

edges⟦([**x** ↦ (**any** ...)] ...)⟧ = ∪⟦*edges*⟦([**x** ↦ (**any** ...)])⟧, ...⟧

THE *transitive-closure* meta-function returns the transitive closure of the edges in the graph:[62]

[62] The fixpoint of the transitive relation: $a \mapsto b,\ b \mapsto c \Rightarrow a \mapsto c$.

transitive-closure : ([**x** ↦ (_ ...)] ...) → ([**x** ↦ (_ ...)] ...)

transitive-closure⟦**any**⟧ = *transitive-closure**⟦**any**, *transitive-closure*/step*⟦**any**⟧⟧

*transitive-closure** : ([**x** ↦ (_ ...)] ...) ([**x** ↦ (_ ...)] ...) → ([**x** ↦ (_ ...)] ...)

*transitive-closure**⟦**any**, **any**⟧ = **any**

*transitive-closure**⟦$\mathbf{any}_1$, $\mathbf{any}_2$⟧ = *transitive-closure**⟦$\mathbf{any}_2$, *transitive-closure*/step*⟦$\mathbf{any}_2$⟧⟧

transitive-closure/step* : ([**x** ↦ (_ ...)] ...) → ([**x** ↦ (_ ...)] ...)

transitive-closure/step*⟦**any**⟧ = *transitive-closure*/step/step*⟦**any**, **any**⟧

transitive-closure/step/step* : ([**x** ↦ (**any** ...)] ...) ([**x** ↦ (_ ...)] ...) → ([**x** ↦ (_ ...)] ...)

transitive-closure/step/step*⟦((), **any**⟧ = ()

transitive-closure/step/step*⟦([$\mathbf{x}_1$ ↦ ($\mathbf{any}_1$ ...)] [$\mathbf{x}_2$ ↦ ($\mathbf{any}_2$ ...)] ...), $\mathbf{any}_3$⟧ = ∪⟦([$\mathbf{x}_1$ ↦ ∪⟦($\mathbf{any}_1$ ...), *lookup**⟦$\mathbf{any}_1$, $\mathbf{any}_3$, ()⟧, ...⟧]),
transitive-closure/step/step*⟦([$\mathbf{x}_2$ ↦ ($\mathbf{any}_2$ ...)] ...), $\mathbf{any}_3$⟧⟧

A.5 Syntax

THE FOLLOWING are auxiliary definitions for the abstract syntax (§ 3):

THE $x/\tau{\to}x$ metafunction extracts the variable from a variable assertion:

x/τ->*x* : **x**/**τ** → **x**

x/τ->*x*⟦**x**⟧ = **x**

x/τ->*x*⟦[**x** : **τ**]⟧ = *x*/τ->*x*⟦**x**⟧

THE *simplify-opn* meta-function normalizes a (non-gramatical) use of an **opn** nonterminal with only one operand:

simplify-opn : _ → **e**

simplify-opn⟦(**opn e**)⟧ = **e**

simplify-opn⟦**any**⟧ = **any**

A.6 Booleans

THE FOLLOWING are constructive definitions of boolean (logic) operators:

THE ¬ meta-function returns the opposite boolean:

$\neg$: **boolean** → **boolean**

$\neg[\![\#t]\!] = \#f$

$\neg[\![\#f]\!] = \#t$

THE $\wedge$ meta-function returns true (#t) if and only if all the arguments are true:

$\wedge$: **boolean** ... → **boolean**

$\wedge[\![\#t, ...]\!] = \#t$

$\wedge[\![\textbf{any}, ...]\!] = \#f$

THE *if* meta-function returns the second argument if the first is true (#t), otherwise it returns the third argument:

if : **boolean** _ _ → _

if$[\![\#t, \textbf{any}_1, \textbf{any}_2]\!] = \textbf{any}_1$

if$[\![\#f, \textbf{any}_1, \textbf{any}_2]\!] = \textbf{any}_2$

A.7 Arithmetic

THE FOLLOWING operations perform arithmetic in SDL:

THE *merge-*$\times$ meta-function returns the abstract syntax for multiplying (×) the arguments, avoiding superfluously nested operands:

merge-$\times$: **e** **e** → **e**

merge-$\times[\![(\times \mathbf{e}_1 ...), (\times \mathbf{e}_2 ...)]\!] = (\times \mathbf{e}_1 ... \mathbf{e}_2 ...)$

merge-$\times[\![\mathbf{e}_1, (\times \mathbf{e}_2 ...)]\!] = (\times \mathbf{e}_1 \mathbf{e}_2 ...)$

merge-$\times[\![(\times \mathbf{e}_1 ...), \mathbf{e}_2]\!] = (\times \mathbf{e}_1 ... \mathbf{e}_2)$

merge-$\times[\![\mathbf{e}_1, \mathbf{e}_2]\!] = (\times \mathbf{e}_1 \mathbf{e}_2)$

THE *invert* meta-function returns the abstract syntax for the inverse of the given argument, avoiding nested inverses:

invert : **e** → **e**

invert$[\![(\times \mathbf{e} ...)]\!] = (\times \textit{invert}[\![\mathbf{e}]\!] ...)$

invert$[\![(\uparrow \mathbf{e}_{base} \text{-}1)]\!] = \mathbf{e}_{base}$

invert$[\![(\uparrow \mathbf{e}_{base} (\text{-} \mathbf{e}_{exponent}))]\!] = (\uparrow \mathbf{e}_{base} \mathbf{e}_{exponent})$

invert$[\![(\uparrow \mathbf{e}_{base} \mathbf{e}_{exponent})]\!] = (\uparrow \mathbf{e}_{base} (\text{-} \mathbf{e}_{exponent}))$

invert$[\![\mathbf{e}_{base}]\!] = (\uparrow \mathbf{e}_{base} \text{-}1)$

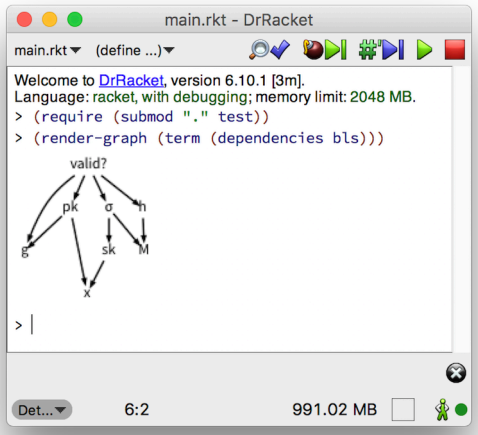
B Visualization

WE DESIGNED SDL and most operations on it directly in Redex, avoiding escaping to Racket with unquote (`,`), which makes the model self-contained and complete. But that power is still available to us, and it is advantageous to explore it for some use cases. In this

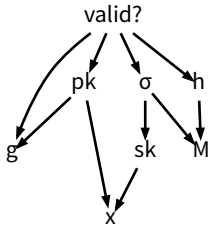
section we introduce an example of this: a visualization technique for the graphs generated by some analyses passes, including Dependence Analysis (§ 8).

The graphs are represented as extended language constructs, via an encoding included in a language extension (§ 4). But, from Racket's standpoint, language constructs are regular data structures (S-expressions); we can access them and perform arbitrary computations. We compile the graph into a Graphviz[63] representation and call out to the tool.

The main use of this visualization is to aid debugging the specification. To this end, we integrated it in the programming environment, DrRacket[64]:



The whole tool was implemented in less than thirty minutes and twenty lines of code, demonstrating the practicality of this technique. Besides this simple example, the tight integration between the Redex model and the Racket ecosystem hosting it provide a promising extension surface. When the rewriting techniques introduced in this report (§ 10) do not suffice for an application, escaping to Racket can solve the issue. We anticipate this will be useful, for example, to let the transformation passes interact with SMT solvers to find solutions for constraints introduced by data-flow dependencies.



Example output: BLS dependency graph.

[63] Ellson et al. 2001.

[64] Findler et al. 2002.