A Method to Translate Order-Sorted Algebras to Many-Sorted Algebras

Liyi Li and Elsa Gunter
Department of Computer Science,
University of Illinois at Urbana-Champaign
{liyili2,egunter}@illinois.edu

Order-sorted algebras and many sorted algebras are classical ideas with many different implementations and applications. In order to marry these two worlds, in this paper, we propose an algorithm to translate an *strictly sensible* order-sorted algebra to a many-sorted one in a restricted domain by requiring the order-sorted algebra to have a restricted format. The key idea of the translation is to add an equivalence relation called *core equality* to the translated many-sorted algebras. By defining this relation, we reduce the complexity in translating an *strictly sensible* order-sorted algebra to a many-sorted one, increase the translated many-sorted algebra equations by less than a linear factor and keep the number of rewrite rules in the algebra in the same amount. We then prove the order-sorted algebra and the many-sorted algebra are bisimilar. We believe that our translation benefits the translations of order-sorted specifications in languages such as $\mathbb{K}$ or Maude to many-sorted systems in theorem provers such as Isabelle/HOL or Coq, which allows users to prove theorems about large and popular language specifications.

1 Motivation

Currently, order-sorted algebras are used widely in defining specifications and programs. Maude [3] and $\mathbb{K}$ [20] are successful programming languages for defining order-sorted algebras. The specifications of a lot of popular programming languages, such as Java [2], Javascript [18], PHP [9], C [8, 12], LLVM and Python semantics, have been defined in $\mathbb{K}$ in a form of order-sorted algebra. Experience shows that order-sorted algebras allow users to define specifications easily.

On the other hand, many-sorted algebras also have wide usage. Many people define pieces of popular programming languages such as C, Java, LLVM and Python in forms of many-sorted algebras. For example, people define specifications based on many-sorted algebras in some interactive theorem provers, such as Isabelle/HOL[19] and Coq[5], where people commonly use their many-sorted type theories to prove properties about language specifications.

In order to connect these two worlds, especially to connect the existing programming language semantic specifications defined in the order-sorted algebra $\mathbb{K}$ with the traditional theorem provers such as Isabelle/HOL and Coq, the key is to discover a way to translate an order-sorted algebra into a many-sorted algebra. The reason we want to do this is to use the theorem proving engines to develop theories about specifications defined in the order-sorted world. Please note that the syntax of the specification that we are interested in translating from a order-sorted form to a many-sorted form is an abstract syntax not a concrete syntax of a language. Even though users are allowed to defined mixfix syntax in order-sorted

Acknowledgments. This material is based upon work supported in part by NSF Grant 0917218. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Submitted to:
WPTE 2017

programming languages such as $\mathbb{K}$ or Maude, they are still representing the abstract syntax and not the concrete syntax of a specification because the mixfix syntax forms are just syntactic sugars $\mathbb{K}$ and Maude use to write abstract syntax for a specification. For example, both $\mathbb{K}$ and Maude do not allow users to create over loaded constants.

To the best of our knowledge, the most recent and relevant work on defining a translation mechanism is that of Meseguer and Skeirik[16], who created an algorithm to translate an order-sorted algebra to a many-sorted one. However, this algorithm deals with the most general cases, so it adds more sorts and rewrite rules than are needed in more restricted cases. In some order-sorted algebras, if some rewrite rules have many sorts and subsorts, it can cause their algorithm to generate rewrite rules exponentially. Even though the chance of this extreme situation is rare, for a normal order-sorted algebra, their algorithm squares or cubes the number of equations and rewrite rules when creating a many-sorted algebra image, which is not desirable.

Our main goal is to connect the world of people defining language specifications using order-sorted algebras to that of people using theorem provers to develop theories about language specifications by using many-sorted algebras. In order to succeed, our translation of an order-sorted algebra must be understood by the people who are using the theorem provers. Making a many-sorted algebra with relatively the same amount of rewrite rules would significantly reduce the users' efforts to understand the translated language specifications. That is the reason for us to present a way to translate an interesting subset of order-sorted algebras into many-sorted algebras with increasing the number of the equations by less than a linear factor and keeping the number of rewrite rules in the same amount.

The basic idea of our algorithm is to view the subsort relation $s \leq s'$ defined in an order-sorted algebra as the implicit coercion of a term in the subsort s to a term in the supersort s'. Then, we borrow the idea of constructors, as a way of explicit coercion, from other functional programming languages, such as Standard ML [17]. We add an explicit coercion with a constructor for each subsort relation and view these subsort relations as unary operators in the translated many-sorted algebra. After that, we add a new equivalence relation for operators, which we call core equality. Core equality allows users to equalize two terms as long as their core parts (not counting the subsort unary operator parts) are the same. By this translation process, we are able to translate a valuable subset of order-sorted algebras into many-sorted ones. Specifically, we are able to translate all those valuable language specifications in $\mathbb{K}$ mentioned above into ones in Isabelle/HOL.

2 The Scope of the Solution

In this section, we describe the exact problem that we want to solve. The basic idea is to find a translation function tr to translate an order-sorted algebra to a many-sorted one and preserve the meaning of the former one in the latter one. We first define term algebras for many-sorted algebra and order sorted algebra in Definition 2.1 and 2.2, respectively. A term algebra is a trivial algebra that defines the terms allowed in an algebra without variables.

Definition 2.1. A *sorted ground term algebra* T_Σ is a tuple of (S, Φ, Σ), where S is a set of sorts, Φ is a finite set of constructors, and Σ is the set of all operators in the system, where an operator is of the form of $f : s_1 \times ... \times s_n \to s$, where f is the constructor in set Φ, $s_1, ..., s_n$ is a list of argument sorts and s is the target sort. Sorts $s_1, ..., s_n$ and s are elements of set S. We call (S, Φ, Σ) the *signature* of the term algebra. Sometimes we use Σ to refer to the signature. The set of terms T_Σ is equal to $\cup_{s \in S}(T_{\Sigma,s})$, where the sets $(T_{\Sigma,s})$ are mutually defined by:

(1) For each operator $a : nil \rightarrow s \in \Sigma$, the constructor $a \in T_{\Sigma,s}$, where nil means that the argument sort list of the operator is an empty list.

(2) For each non-zero arity operator $f : w \rightarrow s \in \Sigma$, where $w = s_1 \times ... \times s_n$ and $n > 0$, and for each $(t_1, ...t_n) \in T_{\Sigma,s_1} \times ... \times T_{\Sigma,s_n}$, and the term $f(t_1, ..., t_n) \in T_{\Sigma,s}$.

Definition 2.2. An *order-sorted ground term algebra* T_Σ is a tuple (S, O, Φ, Σ), where (S, Φ, Σ) is a *sorted ground term algebra* T'_Σ. The set O is a set of pairs of sorts, such that the reflexive and transitive closure $\leq$ forms a partial order, which means that O cannot have cycles if we view the pairs of O as defining a directed graph. The poset $(S, \leq)$ represents the subsort relations of the system. Sometimes we use Σ to refer to the signature. Terms allowed in T_Σ can follow the rules:

(1) $T'_\Sigma \subseteq T_\Sigma$.

(2) If $s \leq s'$, then $T_{\Sigma,s} \subseteq T_{\Sigma,s'}$.

(3) For each non-zero arity operator $f : w \rightarrow s \in \Sigma$, where $w = s_1 \times ... \times s_n$ and $n > 0$, the domain of the argument list of the constructor f is $T_{\Sigma,s_1} \times ... \times T_{\Sigma,s_n}$, where T_{Σ,s_i} include all terms in subsorts of s_i.

$$
\begin{array}{ll}
S: & \{\texttt{nat}, \texttt{int}, \texttt{AExp}, \texttt{Id}, \texttt{bool}, \texttt{BExp}, \texttt{Block}, \texttt{Stmt}, \texttt{Map}, \texttt{Pgm}\} \\
O: & \{\texttt{nat} < \texttt{int}, \texttt{int} < \texttt{AExp},\ \texttt{Id} < \texttt{AExp},\ \texttt{bool} < \texttt{BExp},\ \texttt{Block} < \texttt{Stmt}\} \\
\Phi: & \{\boldsymbol{v},\ \boldsymbol{true},\ \boldsymbol{false},\ 0,\ \boldsymbol{s},\ +,\ -,\ <=,\ \{\},\ \{\_\},\ \_=\_;,\ \_\_,\ \boldsymbol{if\_else},\ \_,\_,\ \boldsymbol{.Map}, \\
 & \boldsymbol{guess},\ <\_,\_>,\ \_[\_/\_],\ \_\mapsto\_\} \\
\Sigma: & \{\boldsymbol{true} :\rightarrow \texttt{bool},\ \ \boldsymbol{false} :\rightarrow \texttt{bool},\ \ 0 :\rightarrow \texttt{nat},\ \ \boldsymbol{s} : \texttt{nat} \rightarrow \texttt{nat},\ \ - : \texttt{int} \rightarrow \texttt{int},\ \ - : \texttt{nat} \rightarrow \texttt{int}, \\
 & + : \texttt{AExp} * \texttt{AExp} \rightarrow \texttt{AExp},\ \ + : \texttt{nat} * \texttt{nat} \rightarrow \texttt{AExp},\ \ + : \texttt{int} * \texttt{int} \rightarrow \texttt{AExp},\ \ \{\} :\rightarrow \texttt{Block}, \\
 & <=: \texttt{AExp} * \texttt{AExp} \rightarrow \texttt{BExp},\ \ - : \texttt{BExp} \rightarrow \texttt{BExp},\ \ - : \texttt{bool} \rightarrow \texttt{BExp},\ \ + : \texttt{bool} * \texttt{bool} \rightarrow \texttt{BExp}, \\
 & \boldsymbol{v} : \texttt{nat} \rightarrow \texttt{Id},\ \ \{\_\} : \texttt{Stmt} \rightarrow \texttt{Block},\ \ \_=\_; : \texttt{Id} * \texttt{AExp} \rightarrow \texttt{Stmt},\ \ \_\_ : \texttt{Stmt} * \texttt{Stmt} \rightarrow \texttt{Stmt}, \\
 & \boldsymbol{if\_else} : \texttt{BExp} * \texttt{Block} * \texttt{Block} \rightarrow \texttt{Stmt},\ \ \boldsymbol{guess} : \texttt{Id} \rightarrow \texttt{int}, \\
 & <\_,\_>: \texttt{Map} * \texttt{Stmt} \rightarrow \texttt{Pgm},\ \ \_,\_ : \texttt{Map} * \texttt{Map} \rightarrow \texttt{Map},\ \ \boldsymbol{.Map} :\rightarrow \texttt{Map},\ \ + : \texttt{BExp} * \texttt{BExp} \rightarrow \texttt{BExp}, \\
 & \_[\_/\_] : \texttt{Map} * \texttt{int} * \texttt{Id} \rightarrow \texttt{Map},\ \ \_\mapsto\_ : \texttt{Id} * \texttt{int} \rightarrow \texttt{Map}\}
\end{array}
$$

Figure 1: IMP Signature

In Figure 1, we show the order-sorted signature of IMP and list the sets of S, O, Φ and Σ accordingly. Based on the signature, the order-sorted ground term algebra T_Σ can be generated by the rules in Definition 2.2. If we cut the O set, the signature becomes a sorted signature, and we can generate the sorted ground term algebra T_Σ by the rules in Definition 2.1. In our version of IMP language, we assume that all identifiers in a given term has been initialized. We do not provide semantics for how to lookup the value for an identifier. Instead, we assume that there is a ***guess*** function that will guess a value for an identifier, which happens to be the same as the value previously defined for the identifier. The reason is that describing a lookup function for an identifier requires a lot more operators, equations and rules for the IMP language, but we cares about the mechanism of translating order-sorted algebras into many-sorted algebras instead of the semantics of IMP. Finally, we use $-$ operator to mean both an integer negative sign and a negation of a boolean formula, as well as $+$ to mean both an arithmetic addition operator and a conjunctive boolean operator, in order to show how we deal with overloaded operators.

Based on the ground term algebra T_Σ, we can talk about the terms with variables as $T_\Sigma(X)$. Given a term with variables $t(X) \in T_\Sigma(X)$, where all variable in $t(X)$ are contained in X, term $t \in T_\Sigma$ is an *instance* of $t(X)$ if there exists a substitution h mapping X to T_Σ such that t is the result of replacing each variable x in $t(X)$ by $h(x)$. Every variable in a term in $T_\Sigma(X)$ is represented by a name. Even though we refer to T_Σ and $T_\Sigma(X)$ as term algebras in both many-sorted algebras and order sorted algebras, They are sorted term algebras in many-sorted world and order sorted term algebras in order sorted world. It is worth noting that, while Σ contains sort information, T_Σ and $T_\Sigma(X)$ do not. A mapping function $x : s$ maps a variable x to a sort s representing the target sort of x. We now define a many-sorted algebra and an order sorted-algebra in Definitions 2.3 and 2.4, respectively.

Definition 2.3. A *many-sorted algebra B* is a tuple (S,Φ,Σ,E,R), where S is a set of sorts, Φ is the finite set of constructors allowed in the system, Σ represents all operators in the system and (S,Φ,Σ) is the many-sorted signature, which we refer to as Σ. The equation set E is a set of pairs of terms in $T_\Sigma(X)$. and partitions the terms in B, T_Σ, into equivalence classes, denoted $T_{(\Sigma,E)}$. The terms allowed to construct each equation in E are in sorted term algebra $T_\Sigma(X)$, while the equations are applied on the terms in the sorted ground term algebra T_Σ. We introduce the quotient structure $T_{(\Sigma,E)}$, which we call terms T_Σ modulo equations E. For two terms t and t' in T_Σ, if we can prove they are equal through the equations E, we say these two terms are equivalent modulo E, which partitions T_Σ into different equivalence classes and forms $T_{(\Sigma,E)}$. A set of rewrite rules R define the semantics of system B. The rule set R is a set of pairs of terms in $T_\Sigma(X)$, while the rules are applied on the terms in $T_{(\Sigma,E)}$. A rule $r \in R$ is applied to a class $c \in T_{(\Sigma,E)}$, such that $c \longrightarrow_r c'$. The transition $c \longrightarrow_r c'$ means that for $t(X)$ as the left hand side and $t'(X)$ as the right hand side of rule r, there is a substitution h mapping X to T_Σ such that t and t' are the result of replacing each variable x in $t(X)$ and $t'(X)$ by $h(x)$ and $t \in c$ and $t' \in c'$, respectively. The rule r generates a endomorphic relation, and applications of rules are closed under applications of constructors.

Definition 2.4. An *order-sorted algebra A* is a tuple (S,O,Φ,Σ,E,R), where (S,Φ,Σ) is a many-sorted signature, and O is a set of pairs of sorts, such that the reflexive and transitive closure $\leq$ forms a partial order. The poset $(S,\leq)$ represents the subsort relations of the system. We call (S,O,Φ,Σ) the signature of the system, which we refer to as Σ. The terms allowed to construct each equation in E are in order-sorted term algebra $T_\Sigma(X)$, while the equations are applied on the terms in order-sorted ground term algebra T_Σ. A set of rewrite rules R define the semantics of system B. The rule set R is a set of pairs of terms in $T_\Sigma(X)$, while the rules are applied on the terms in $T_{(\Sigma,E)}$. The two elements of a pair in E are required to have the same sort, while for any pair (c,c') in R, the sort of c' is a subsort of the sort of c. This property is called *sort decreasing*. We introduce the quotient structure $T_{(\Sigma,E)}$, which we call terms T_Σ modulo equations E. For two terms t and t' in T_Σ, if we can prove they are equal through the equations E, we say these two terms are equivalent modulo E, which partitions T_Σ into different equivalence classes and forms $T_{(\Sigma,E)}$. A set of rewrite rules R define the semantics of system B. The rule set R is a set of pairs of terms in the order-sorted term algebra $T_\Sigma(X)$, while the rules are applied on the terms in $T_{(\Sigma,E)}$. A rule $r \in R$ is applied to a class $c \in T_{(\Sigma,E)}$, such that $c \longrightarrow_r c'$. The transition $c \longrightarrow_r c'$ means that for $t(X)$ as the left hand side and $t'(X)$ as the right hand side of rule r, there is a substitution h mapping X to T_Σ such that t and t' are the result of replacing each variable x in $t(X)$ and $t'(X)$ by $h(x)$ and $t \in c$ and $t' \in c'$, respectively. The rule r generates a endomorphic relation, and applications of rules are closed under applications of constructors.

The $\leq$ relation can be viewed as a directed graph where each relation is an edge. The graph may have different connected components. For any two sorts in a connected component in an order-sorted algebra, we require there is a unique top *supersort* of them.

In Figure 2, we show the equations and rules for the order-sorted algebra IMP. With the information in Figure 1, This information constructs a well-defined order-sorted algebra. A many-sorted algebra is similar to this one with more restrictions. For example, the left hand side and right hand side of a rule need to be sort equivalent in a many-sorted algebra. We also cannot write an equation to match against terms having subsorts of the left hand side of the equation. In order to write an equation for $+$ operator, we either deletes some overloaded $+$ operators, or we need to write three versions: one for $+$ with argument sorts $\texttt{AExp} * \texttt{AExp}$, one for it with argument sorts $\texttt{int} * \texttt{int}$ and one for argument sorts $\texttt{nat} * \texttt{nat}$.

$$
\begin{array}{ll}
E: & \{0+A:\texttt{AExp}=A:\texttt{AExp},\ \boldsymbol{s}(A:\texttt{nat})+B:\texttt{nat}=A:\texttt{nat}+\boldsymbol{s}(B:\texttt{nat}),\ --A:\texttt{int}=A:\texttt{int},\\
& A:\texttt{AExp}+B:\texttt{AExp}=B:\texttt{AExp}+A:\texttt{AExp},\\
& \boldsymbol{s}(A:\texttt{nat})+-\boldsymbol{s}(B:\texttt{nat})=A:\texttt{nat}+B:\texttt{nat},\ \boldsymbol{true}+A:\texttt{BExp}=A;\texttt{BExp},\\
& A:\texttt{BExp}+B:\texttt{BExp}=B:\texttt{BExp}+A:\texttt{BExp},\ \_,\_(A:\texttt{Map},B:\texttt{Map})=\_,\_(B:\texttt{Map},A:\texttt{Map}),\\
& \boldsymbol{s}(A:\texttt{nat})<=B:\texttt{AExp}=0<=B:\texttt{AExp}+-\boldsymbol{s}(A:\texttt{nat}),\ \_,\_(A:\texttt{Map},\boldsymbol{.Map})=A:\texttt{Map},\\
& -\boldsymbol{s}(A:\texttt{nat})<=B:\texttt{AExp}=0<=B:\texttt{AExp}+\boldsymbol{s}(A:\texttt{nat}),\\
& \_,\_(A:\texttt{Map},\_,\_(B:\texttt{Map},C:\texttt{Map}))=\_,\_(\_,\_(A:\texttt{Map},B:\texttt{Map}),C:\texttt{Map}),\\
& \_[\_/\_](\boldsymbol{.Map},A:\texttt{int},B:\texttt{Id})=B:\texttt{Id}\mapsto A:\texttt{int},\\
& \_[\_/\_](\_,\_(A:\texttt{Id}\mapsto B:\texttt{int},C:\texttt{Map}),D:\texttt{int},A:\texttt{Id})=\_,\_(A:\texttt{Id}\mapsto D:\texttt{int},C:\texttt{Map}),\\
& \_[\_/\_](\_,\_(A:\texttt{Id}\mapsto B:\texttt{int},C:\texttt{Map}),D:\texttt{int},E:\texttt{Id})\\
& \qquad =\_,\_(A:\texttt{Id}\mapsto B:\texttt{int},\_[\_/\_](C:\texttt{Map},D:\texttt{int},E:\texttt{Id})),\ \_\_(\boldsymbol{\{\}},A:\texttt{Stmt})=A:\texttt{Stmt},\\
& \_\_(\boldsymbol{\{\_\}}(A:\texttt{Stmt}),B:\texttt{Stmt})=\_\_(A:\texttt{Stmt},B:\texttt{Stmt})\ \}\\
R: & \{-0\Rightarrow 0,\ A:\texttt{AExp}+\boldsymbol{v}(B:\texttt{Id})\Rightarrow A:\texttt{AExp}+\boldsymbol{guess}(B:\texttt{Id}),\ -\boldsymbol{true}\Rightarrow\boldsymbol{false},\ -\boldsymbol{false}\Rightarrow\boldsymbol{true},\\
& A:\texttt{AExp}<=\boldsymbol{v}(B:\texttt{Id})\Rightarrow A:\texttt{AExp}<=\boldsymbol{guess}(B:\texttt{Id}),\ 0<=A:\texttt{nat}\Rightarrow\boldsymbol{true},\\
& 0<=-\boldsymbol{s}(A:\texttt{nat})\Rightarrow\boldsymbol{false},\ \boldsymbol{v}(A:\texttt{Id})<=B:\texttt{AExp}\Rightarrow\boldsymbol{guess}(A:\texttt{Id})<=B:\texttt{AExp},\\
& <\_,\_>(A:\texttt{Map},\_\_(\boldsymbol{if\_else}(\boldsymbol{false},B:\texttt{Block},C:\texttt{Block}),D:\texttt{Stmt}))\\
& \qquad \Rightarrow <\_,\_>(A:\texttt{Map},\_\_(C:\texttt{Block},D:\texttt{Stmt})),\\
& <\_,\_>(A:\texttt{Map},\_\_(\boldsymbol{if\_else}(\boldsymbol{true},B:\texttt{Block},C:\texttt{Block}),D:\texttt{Stmt}))\\
& \qquad \Rightarrow <\_,\_>(A:\texttt{Map},\_\_(B:\texttt{Block},D:\texttt{Stmt})),\\
& <\_,\_>(A:\texttt{Map},\_\_(\_=\_;(B:\texttt{Id},C:\texttt{int}),D:\texttt{Stmt}))\\
& \qquad \Rightarrow <\_,\_>(\_[\_/\_](A:\texttt{Map},C:\texttt{int},B:\texttt{Id}),D:\texttt{Stmt}),\\
& -A:\texttt{int}+-B:\texttt{int}\Rightarrow-(A:\texttt{int}+B:\texttt{int}),\ \boldsymbol{false}+A:\texttt{BExp}\Rightarrow\boldsymbol{false}\ \}
\end{array}
$$

Figure 2: IMP Order-Sorted Equations and Rules

In a many-sorted algebra and order-sorted algebra, even though the terms that are used to construct an equation or a rule is in the form of $T_\Sigma(X)$, they are representatives of equivalent classes in $T_{(\Sigma,E)}$. One thing to keep in mind is that we are defining algebras in this paper, not transition systems. The rewrite rules in an algebra can be applied to any subterms of a given term, not only to its top-most operator. This idea is similar to the rewrite rules in Rewriting Logic [13]. Based on the order-sorted algebra definition, the only input restriction of our translation function tr is that the order-sorted algebra A should be, not just *sensible*, but *strictly sensible*. The former term is defined in Definition 2.6, while the latter is defined in Definition 2.7. One thing about *overloaded operators* (two operators having the same constructor) in an algebra is that if the two overloaded operators f and f' have argument sorts that have no common supersorts, we treat them as different operators since they can be easily distinguished by combining the constructor and the list of argument sorts.

Definition 2.5. We define two overloaded operators f and f' to be *argument compatible*, if they have the same arities, and f has argument sorts $s_1,...,s_n$, and f' has argument sorts $s'_1,...,s'_n$, and $s_i \equiv_\leq s'_i$ for $i = 1,...,n$, where $\equiv_\leq$ means that the two given sorts have a common supersort.

Definition 2.6. (Goguen and Meseguer [11]) An order-sorted algebra is *sensible*, if for any pair of argument compatible constructors f and f' with target sorts s and s', respectively, we have $s \equiv_\leq s'$.

Definition 2.7. An order-sorted algebra is *strictly sensible* if:

(1) Whenever there are two argument compatible operators f and f' with target sorts s and s', respectively, then we have $s = s'$. We then call the order-sorted algebra being *(strong sensible)*. It is worth

noting that a strong sensible algebra cannot have overloaded constant operators.

(2) For each operator f, there exists an operator $f' : s_1 \times ... \times s_n \to s$, such that for every operator f'' being argument compatible with f, f' is argument compatible with f'', and if f'' have argument sorts $s'_1 \times ... \times s'_n$, then $s'_i \leq s_i$ for all $i = 1, ..., n$. We then all the order-sorted algebra being *(maximal argument-bounding)*.

The order-sorted algebra in Figure 1 and 2 is strictly sensible, but if we change the operator $+ :$ $\texttt{nat} * \texttt{nat} \to \texttt{AExp}$ to be $+ : \texttt{nat} * \texttt{nat} \to \texttt{nat}$, the algebra becomes sensible not strictly sensible. The second condition of the strictly sensible definition is not necessary, and it ensures that the translated many-sorted algebra from a strictly sensible order-sorted algebra are bi-simulated. Without the condition, the translated many-sorted algebra simulates the order-sorted algebra. The first condition is the key distinction between a sensible order-sorted algebra and a strictly sensible order-sorted algebra. We rule out the possibility for users to define overloaded operator pairs like $+ : \texttt{AExp} * \texttt{AExp} \to \texttt{AExp}$ and $+ : \texttt{nat} * \texttt{nat} \to \texttt{nat}$.

The reason that we are willing to accept the strictly sensible restriction is that users only need the limited world in defining language specifications from scratch. This is a real restriction that will affect some situations, because without the restriction of strictly sensible, we can define two overloaded $+$ operators $+ : \texttt{int} * \texttt{int} \to \texttt{int}$ and $+ : \texttt{nat} * \texttt{nat} \to \texttt{nat}$, where int and nat have a subsort relation. However, there are no operators of this kind in the order-sorted specifications of C [8], PHP [9], JavaScript[18], and Java[2] in $\mathbb{K}$. In addition, the operators, such as $+ : \texttt{int} * \texttt{nat} \to \texttt{int}$ and $+ : \texttt{nat} * \texttt{int} \to \texttt{nat}$, are usually defined as different operators with different names by users. Even though we are able to solve them easily by adding more rules and creating more sorts, such as the algorithm of Meseguer and Skeirik[16] does, we do not want to take that approach because the whole point of the translation is to have a many-sorted algebra that is concise enough for users to use and read the translated language specifications. Squaring or cubing the size of the rewrite rules is quite undesirable.

Now, we can formally state the properties of the translation function tr to be: given an strictly sensible order-sorted algebra A and a translation function tr applied on A, we have a many-sorted algebra B such that $B = tr(A)$, and for any rule r_A in A, if term t_A in A can be transitioned to t'_A through rule r_A, such that $t_A \longrightarrow_{r_A} t'_A$, then we have $tr(r_A)$ is a rule in B and $tr(t_A) \longrightarrow_{tr(r_A)} tr(t'_A)$. The output of our translation is a many-sorted algebra: B, where the rewrite rules of A and B have the above relation.

3 Translation and Proofs

In this section, a description of the translation function tr and some theorems about it are given. For a given order-sorted algebra A with $(S, O, \Phi, \Sigma, E, R)$, we do not need to translate the sort set S because our translation does not change sorts at all. We eliminate the relation O, and we have the functions tr_Σ, tr_E and tr_R for translating operator definitions, equations and rewrite rules.

Translating Operators. Operators are translated in two steps, such that $tr_\Sigma = tr^\#_\Sigma \circ tr'_\Sigma$. The first step tr'_Σ is to find a maximal argument-bounding operator f for every operator f'. Since our strictly sensible assumptions require any pair of argument compatible operators f' and f'' to have the same target sort, we restrict the nature of the argument sorts in these overloaded operators by picking its maximal argument-bounding operator f as a representative for any argument compatible operator f'. We then eliminate the operator f' if f is different from f'. Hence, if $\Sigma' = tr'_\Sigma$, then Σ' has fewer operators than Σ and for every overloaded operator set, whose elements are argument compatible, Σ' picks exactly one representative operator for it. If the overloaded operators have no compatible arguments, then we distinguish them by picking different constructors in the translated many-sorted algebra. In the order-sorted algebra in Figure 1 and 2, there are five different overloaded operators for $+$ constructor, where $+ : \texttt{AExp} * \texttt{AExp} \to \texttt{AExp}$, $+ : \texttt{nat} * \texttt{nat} \to \texttt{AExp}$ and $+ : \texttt{int} * \texttt{int} \to \texttt{AExp}$ are argument compatible, while $+ : \texttt{bool} * \texttt{bool} \to \texttt{BExp}$ and $+ : \texttt{BExp} * \texttt{BExp} \to \texttt{BExp}$ are also argument compatible. These two groups of $+$ operators are not argument compatible cross groups. When translating these operators, we first pick $+ : \texttt{AExp} * \texttt{AExp} \to \texttt{AExp}$ and $+ : \texttt{BExp} * \texttt{BExp} \to \texttt{BExp}$ as the representatives for the first and second group, then we change the name of the first one to ***+AExp*** and the second one to ***+BExp*** to avoid conflict in constructor names.

The translation $tr^{\#}_{\Sigma}$ translates a given Σ' by adding operators. For each pair defined in set O as (s, s'), which is a subsort relation $s \leq s'$, we create one more unary operator **Cast_s_to_s'** : $s \to s'$ that does not appear in Σ'. This operator has argument sort s and target sort s'. The result signature $\Sigma^{\#} = tr^{\#}_{\Sigma}(\Sigma')$ contains a set of newly generated unary operators that have bijective relation with the pairs in O. When translating the order-sorted algebra in Figure 1 and 2, we add the following unary operators: ***Cast_nat_to_int*** : $\texttt{nat} \to \texttt{int}$, ***Cast_int_to_AExp*** : $\texttt{int} \to \texttt{AExp}$, ***Cast_Id_to_AExp*** : $\texttt{Id} \to \texttt{AExp}$, ***Cast_bool_to_BExp*** : $\texttt{bool} \to \texttt{BExp}$ and ***Cast_Block_to_Stmt*** : $\texttt{Block} \to \texttt{Stmt}$. Similar to the theorems in the signature translation of the paper of Meseguer and Skeirik, we also have the following theorem about the final result $\Sigma^{\#}$. The proof of the theorem about is similar to the one in the paper of Meseguer and Skeirik, and is a direct result of the strictly sensible requirement of an order-sorted algebra and our translation of the operators in the algebra.

Theorem 3.1. Let Σ be an order-sorted signature, $\Sigma^{\#}$ is the translated many-sorted algebra of it. All overloaded operators in $\Sigma^{\#}$ have at least one argument position having distinct argument sorts that have no common supersort in the original order-sorted algebra.

Translating Terms and Equations. We have a function tr_E to translate every equation in E to $E^{\#}$ and also add a set of *core equality* equations to $E^{\#}$. After the translation, the terms in the translated many-sorted algebra forms a term algebra $T_{(\Sigma^{\#}, E^{\#})}$, and $T_{(\Sigma^{\#}, E^{\#})}$ also represents the union of term sets for each sort $s \in S$ as $T_{(\Sigma^{\#}, E^{\#}, s)}$. In the quotient structure $T_{(\Sigma^{\#}, E^{\#})}$, the equivalence classes are partitioned by the combination effects of equations $E^{\#}$ and sorts S.

First, the translation tr_E adds equations to the equation set E to generate $E^{\#}$. The new equations relates to the idea of the core of a term. In order to talk about the core of a term, we first define non-core constructors as the new constructors generated during the translation $tr^{\#}_{\Sigma}$. The core constructors are the constructors of the normal operators of Σ. The core part of a term means the t of term $C_1(...(C_n(t))...)$, where $C_1, ..., C_n$ are unary non-core constructors, and the top most constructor of t is a core constructor. We now show the definition of *core equality*.

Definition 3.1. If there are two lists of unary non-core constructors $C_1, ..., C_n$ and $K_1, ..., K_m$, such that $t = C_1(...(C_n(x))...)$ and $t' = K_1(...(K_m(x))...)$ are well-formed, i.e., the input sort of C_i is equal to the output sort of C_{i+1} for all $i = 1, ..., n-1, ...$ and the input sort of K_j is equal to the output sort of K_{j+1} for all $j = 1, ..., m-1, ...$, as well as C_1 and K_1 has target sort s, C_n and K_m has input sort s', then for each pair of directed paths from s' to s in the graph of $\leq$ in the original order-sorted algebra, i.e., $s' \leq s$, we have a equation $C_1(...(C_n(x : s'))...) = K_1(...(K_m(x : s'))...)$. The congruence closure of all these equations is *core equality*.

Theorem 3.2. Core equality is an equivalence relation.

When translating the order-sorted algebra in Figure 1 and 2, the generated unary operators, ***Cast_nat_to_int*** : $\texttt{nat} \to \texttt{int}$, ***Cast_int_to_AExp*** : $\texttt{int} \to \texttt{AExp}$, ***Cast_Id_to_AExp*** : $\texttt{Id} \to \texttt{AExp}$, ***Cast_bool_to_BExp*** :

`bool` $\rightarrow$ `BExp` and $\boldsymbol{Cast\_Block\_to\_Stmt}$: `Block` $\rightarrow$ `Stmt`, are non-core constructors and operators, while the original operators are core ones. To generate the set of core equality equations for the order-sorted algebra, we have a practical way to do it; that is to examine the $\leq$ relation. For every two nodes in $\leq$, if there are more than one paths from the first node to the second one, we add equations to connect them. In the the order-sorted algebra in Figure 1 and 2, if we have one more sort `real` and two more subsort relations, `nat` $<$ `real` and `real` $<$ `AExp`, then two paths can go from `AExp` to `nat` in $\leq$. We add an equation $\boldsymbol{Cast\_nat\_to\_int}(\boldsymbol{Cast\_int\_to\_AExp}(A:\texttt{nat})) = \boldsymbol{Cast\_nat\_to\_real}(\boldsymbol{Cast\_real\_to\_AExp}(A:\texttt{nat}))$ to $E^{\#}$.

After we have core equality, we can translate terms in T_{Σ} and $T_{\Sigma}(X)$. We define a translation function tr_{term} to translate a term in T_{Σ} and $T_{\Sigma}(X)$ to a term in $T_{\Sigma^{\#}}$ and $T_{\Sigma^{\#}}(X)$. For every sub-term $f(t_1,...,t_i,...,t_m)$ of a term t in T_{Σ} or $T_{\Sigma}(X)$, if sort of position i in the term is defined with sort s according to the signature Σ, but t_i has sort s' and $s' \leq s$, then we find a list of non-core unary constructors $C_1,...,C_n$ to cast the sub-term to sort s as $f(t_1,...,C_1(...(C_n(t_i))...),...,t_m)$. If $s' = s$, then we do not need to find the constructors. We know that such sequence of unary constructors must exist because the set of non-core unary constructors is bijective with the pairs in O, and $\leq$ is the reflexive and transitive closure of O. If $s' \leq s$, there is a list of pairs in O as $(s',s_1),...,(s_{n-1},s)$. Through the list, s' reaches s. For each pair in the list, we have generated a unary constructor. Hence, the sequence of constructors $C_1,...,C_n$ is exactly the constructors generated for pairs $(s',s_1),...,(s_{n-1},s)$. When translating the order-sorted algebra in Figure 1 and 2, the equation $\boldsymbol{s}(A:\texttt{nat})+B:\texttt{nat} = A:\texttt{nat}+\boldsymbol{s}(B:\texttt{nat})$ is translated to $\boldsymbol{Cast\_nat\_to\_int}(\boldsymbol{Cast\_int\_to\_AExp}(\boldsymbol{s}(A:\texttt{nat}))) + \boldsymbol{Cast\_nat\_to\_int}(\boldsymbol{Cast\_int\_to\_AExp}(B:\texttt{nat})) = \boldsymbol{Cast\_nat\_to\_int}(\boldsymbol{Cast\_int\_to\_AExp}(A:\texttt{nat})) + \boldsymbol{Cast\_nat\_to\_int}(\boldsymbol{Cast\_int\_to\_AExp}(\boldsymbol{s}(B:\texttt{nat})))$.

In defining tr_{term}, for each pair of relation (s,s') in $\leq$, we pick a well-formed constructor sequence $C_1,...,C_n$, such that the target sort of C_1 is s' and the input sort of C_n is s. The sequence defines the way of translating a sub-term t having sort s to a sort s' by constructing $C_1(...(C_n(t))...)$ in tr_{term}. Because of core equality, the choice does not affect the construction of the equivalence classes in $T_{\Sigma^{\#},E^{\#}}$, and not affect the represented equivalence classes by a term in $T_{\Sigma^{\#}}(X)$. We have three theorems about the translation function tr_{term} and term in $T_{\Sigma^{\#},E^{\#}}$ and $T_{\Sigma^{\#}}(X)$.

Theorem 3.3. Let Σ be an order-sorted signature, T_{Σ} be the term algebra of it, E be the equation set of the order-sorted algebra, $T_{\Sigma,E}$ be the terms T_{Σ} modulo equations E, $T_{\Sigma}(X)$ be the terms with variables in the order-sorted algebra, $\Sigma^{\#}$ be the translated many-sorted algebra of signature Σ, $T_{\Sigma^{\#}}$, $E^{\#}$, $T_{\Sigma^{\#},E^{\#}}$ and $T_{\Sigma^{\#}}(X)$ are the corresponding translations of items in the order-sorted algebra, and tr_{term} be the translation function of terms.

(1) If a term t has least sort s in Σ, then its translation t' has the target sort s.

(2) For a term t in $T_{\Sigma,E}$, for any two term translation functions tr_{term} and tr'_{term} having difference in picking different sequences of constructors for pairs in $\leq$, if $c \in T_{\Sigma^{\#},E^{\#}}$ and $tr_{term}(t) \in c$, then $tr'_{term}(t) \in c$.

(3) For a term $t(X)$ in $T_{\Sigma}(X)$, for two translation functions tr_{term} and tr'_{term} having difference in picking different sequences of constructors for pairs in $\leq$, we have two terms $tr_{term}(t(X))$ and $tr'_{term}(t(X))$, for any substitution h mapping X to $T^{\#}_{\Sigma}$ such that t and t' are the result of replacing each variable x in $tr_{term}(t(X))$ and $tr'_{term}(t(X))$ by $h(x)$, if $c \in T_{\Sigma^{\#},E^{\#}}$ and $t \in c$, then $t' \in c$.

Proof. Part (1) is trivial because after we require our operators to be strictly sensible, so any term must have a unique least target sort in the original order-sorted algebra and the target sort is also the target sort of the translated term in $T_{\Sigma^{\#}}$ without converting it to other supersort s' by adding non-core constructors on top of it.

To show (2), if for a term t having sort s', and the translation functions tr_{term} and tr'_{term} cast it into a term in sort s without the need of translating the subterms of t, then the two results terms $tr_{term}(t)$ and

$tr'_{term}(t')$ are trivially in the same equivalent class based on the definition of core equality. s' and s must be the same because the order-sorted definition in this paper requires sort equivalence in two sides of an equation.

If there is a term t having a subterm $f(t_1,...,t_n)$, if the position i of the list $t_1,...,t_n$ has target sort s according to the signature, the term t_i has sort s' and $s' \leq s$, then a given translation function generates well-formed non-core constructor sequences having the form $C_1,...,C_n$ to translate the term t_i. We refer to the number of the non-core constructors in this sequence as n, which is the same as one of the distances between s' and s in O, We induct on maximal numbers of the non-core constructors in each argument position in a term t. If the maximal number of argument non-core constructors is zero, it means that tr_{term} and tr'_{term} do not translate the direct subterms of $f(t_1,...,t_n)$, so any translations on $f(t_1,...,t_n)$ to a target sort s'' generate terms in the same equivalent class. Assuming that when the maximal numbers of non-core constructors are less than k, $tr_{term}(f(tr_{term}(t_1),...,tr_{term}(t_n)))$ and $tr'_{term}(f(tr'_{term}(t_1),...,tr'_{term}(t_n)))$ generate terms in the same equivalent class; if the position i in $f(t_1,...,t_n)$ has sort s, the term t_i has sort s', $s' \leq s$ and the maximal distance between s' and s is $k+1$, if there is only one path from s to reach s', then $tr_{term}(t_i)$ must be the same as $tr'_{term}(t_i)$ since we generate only one non-core constructor for each pair in O. If there are at least two paths, without losing generality, assuming that tr_{term} has the longest path, tr_{term} picks the well-formed sequence $C_1,...,C_{k+1}$ to translate t_i to a term having sort s and tr'_{term} picks the well-formed sequence $K_1,...,K_m$ to translate t_i to a term having sort s, where $m \leq k+1$. Based on the definition of core equality, $C_1(...(C_{k+1}(t_i))...) =_{core} K_1(...(K_m(t_i))...)$, hence, any argument t_i of $f(t_1,...,t_n)$ are translated by tr_{term} and tr'_{term} into terms in the same equivalence class and $f(t_1,...,t_n)$ are also translated by tr_{term} and tr'_{term} into terms in the same equivalence class.

To show (3), the proof basically modifies the proof of part (2) to allow variables in the term and by any substitution on the same variables in two terms t and t' that are generated by tr_{term} and tr'_{term} are in the same equivalence class.

□

Translating Semantic Rules. Translating semantic rules R to $R^{\#}$ is very straight forward and similar to the one in translating equations in E to $E^{\#}$. For each pair $(t(X),t'(X))$ in R, the first step is to apply the term translation on $t(X)$ and $t'(X)$, to be terms in $T_{\Sigma^{\#}}(X)$. Then, we add one rule $(tr_{term}(t(X)),tr_{term}(t'(X)))$ to $R^{\#}$. Since we assume all order-sorted algebra are sort decreasing, the right hand side of a rule have top-most target sort being a subsort of the left hand side of the rule. After they are translated into many-sorted algebras, the two sides of a rule must have the same top-most target sort. We solve this problem by casting the right hand side of a rule to have top-most sort equal to the left hand side. For example, in translating the rule $-0 \Rightarrow 0$ in the order-sorted algebra in Figure 1 and 2, we make a new rule $-0 \Rightarrow$ ***Cast_nat_to_int***(0) in the translated many-sorted algebra.

For a given order-sorted algebra A as (S,O,Φ,Σ,E,R), our translation produces the many-sorted algebra $(S,tr_{\Sigma}(\Sigma),tr_E(E),tr_R(R))$. We believe that our many-sorted algebra maintains a bi-simulation relation as the original order-sorted algebra. The bi-simulation proof is based on structural inductions on the signature (S,O,Φ,Σ) and $(S,tr_{\Sigma}(\Sigma))$.

Theorem 3.4 (Bi-simulation between A and $tr(A)$)**.** Let $(S,tr_{\Sigma}(\Sigma),tr_E(E),tr_R(R))$ be the translated many-sorted algebra of a given order-sorted algebra (S,O,Φ,Σ,E,R), For any r in R and term t in $T_{(\Sigma,E)}$, if $t \longrightarrow_r t'$, then we have $tr_{\Sigma}(t) \longrightarrow_{tr_R(r)} tr_{\Sigma}(t')$. For any p in $T_{(\Sigma^{\#},E^{\#})}$, if $p \longrightarrow_{tr_R(r)} p'$, then there are terms t and t' such that $p = tr_{\Sigma}(t)$, $p' = tr_{\Sigma}(t')$ and $t \longrightarrow_r t'$.

4 Related Work and Conclusion

An order-sorted algebra is a classical idea, which was first systematically introduced into programming language field by Goguen et al. [10]. Many people tried to define rewriting strategies, unifications and equational rules on top of order-sorted algebras [4, 1, 15, 11].

Maude [3] and $\mathbb{K}$ [20] are two implementations of order-sorted algebras as programming languages. They are successful languages since a lot of specifications and applications have been built on top of them, for example, defining popular language specifications, including the semantics of Java [2], Javascript [18], PHP [9], C [8, 12], LLVM and Python, as well as applications [14, 6, 7].

On the other hand, a many sorted algebra is also a classical idea whose logic system has been explored by Wang [21]. Many well-known programming languges such as C, Java, LLVM and Python are based on many-sorted algebras. One of the most prominent and mathematical of programming language specifications, Standard ML by Milner, Tofte, Harper, and Macqueen [17] is based on many sorted algebras. The two famous theorem provers: Isabelle/HOL[19] and Coq[5] are also, which was our motivation to provide a translation from order-sorted algebras into many-sorted ones.

As far as we know, the most recent and unique solution for translating order-sorted algebras into many-sorted ones is by Meseguer and Skeirik[16]. In their translation, by adding possible more sorts, they calculate the least sorts of constructs and put them under corresponding sorts to create the signature of a many-sorted algebra. For any given rule, they add more rules if variables of the rule have subsorts in the original order-sorted algebra. They need to add one more rule for each subsort of a variable in a rule. In dealing with the order-sorted algebra in Figure 1 and 2, to translate the equation $A:\texttt{AExp}+B:\texttt{AExp}=B:\texttt{AExp}+A:\texttt{AExp}$, they generate three different equations: $A:\texttt{nat}+B:\texttt{nat}=B:\texttt{nat}+A:\texttt{nat}$, $A:\texttt{int}+B:\texttt{int}=B:\texttt{int}+A:\texttt{int}$ and $A:\texttt{AExp}+B:\texttt{AExp}=B:\texttt{AExp}+A:\texttt{AExp}$. The original rule involves only one sort `AExp`. if there is a rule involving `AExp`, `BExp` and `Stmt`, which all have subsorts, then the algorithm generates twelve different equations in the translated many-sorted algebra. In fact, if there is a rule or an equation involving n variable having different sorts and each of them have m different subsorts, the algorithm generates m^n different rules or equations in the translated many-sorted algebra. On the other hand, our translation does not change their sorts. We view subsort relations as implicit coercions, while our translation makes them into explicit ones by inserting a constructor for each relation and making the relation into a unary operator in the given order-sorted algebra. We insert a new equational rule named core equality to introduce new partitions on the equivalence classes of the terms allowed in the algebra. Because of these features, our translation keeps relatively same size of rewrite rules in the translated many-sorted algebra and gives users a simpler final description of the language specifications.

Conclusion. In this paper, we proposed an algorithm to translate an order-sorted algebra into a many-sorted one in a restricted domain by requiring the order-sorted algebra to be strictly sensible. The key idea of the translation is to add an equivalence relation called *core equality* to the translated many-sorted algebras. By defining this relation, we reduce the complexity in translating an *strictly sensible* order-sorted algebra to a many-sorted one, increase the translated many-sorted algebra equations by less than a linear factor and keep the number of rewrite rules in the algebra in the same amount. We then prove the order-sorted algebra and the many-sorted algebra are bisimilar (Section 3). We also showed that core equality is indeed an equivalence relation and that our translation bi-simulates the order-sorted and the many-sorted algebras. Along showing our algorithm and theorems, an IMP language is introduced as an exmple of the algorithm. We believe that our translation facilitates translations of order-sorted specifications in $\mathbb{K}$ or Maude into many-sorted systems in Isabelle/HOL or Coq, which will empower users to prove theorems about large and popular language specifications.

References

[1] María Alpuente, Santiago Escobar, Javier Espert & José Meseguer (2014): *A Modular Order-sorted Equational Generalization Algorithm*. Inf. Comput. 235, pp. 98–136, doi:10.1016/j.ic.2014.01.006. Available at `http://dx.doi.org/10.1016/j.ic.2014.01.006`.

[2] Denis Bogdănaş & Grigore Roşu (2015): *K-Java: A Complete Semantics of Java*. In: Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15), ACM, pp. 445–456, doi:http://dx.doi.org/10.1145/2676726.2676982.

[3] Manuel Clavel, Steven Eker, Patrick Lincoln & José Meseguer (2000): *Principles of Maude*. In J. Meseguer, editor: Electronic Notes in Theoretical Computer Science, 4, Elsevier Science Publishers.

[4] Hubert Comon (1990): *Equational formulas in order-sorted algebras*, pp. 674–688. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/BFb0032066. Available at `http://dx.doi.org/10.1007/BFb0032066`.

[5] Pierre Corbineau (2008): *A Declarative Language for the Coq Proof Assistant*, pp. 69–84. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-68103-8_5. Available at `http://dx.doi.org/10.1007/978-3-540-68103-8_5`.

[6] Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln & Carolyn Talcott (2002): *Pathway Logic: Executable Models of Biological Networks*. In: Fourth International Workshop on Rewriting Logic and Its Applications (WRLA'2002), Pisa, Italy, September 19 — 21, 2002, Electronic Notes in Theoretical Computer Science 71, Elsevier. `http://www.elsevier.nl/locate/entcs/volume71.html`.

[7] Steven Eker, José Meseguer & Ambarish Sridharanarayanan (2002): *The Maude LTL Model Checker*. In Fabio Gadducci & Ugo Montanari, editors: Fourth Workshop on Rewriting Logic and its Applications, WRLA '02, Electronic Notes in Theoretical Computer Science 71, Elsevier.

[8] Chucky Ellison & Grigore Rosu (2012): *An Executable Formal Semantics of C with Applications*. In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12), ACM, pp. 533–544, doi:http://doi.acm.org/10.1145/2103656.2103719.

[9] Daniele Filaretti & Sergio Maffeis (2014): *An Executable Formal Semantics of PHP*, pp. 567–592. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-662-44202-9_23. Available at `http://dx.doi.org/10.1007/978-3-662-44202-9_23`.

[10] Joseph A. Goguen, Jean-Pierre Jouannaud & José Meseguer (1985): *Operational Semantics for Order-Sorted Algebra*. In: Proceedings of the 12th Colloquium on Automata, Languages and Programming, Springer-Verlag, London, UK, UK, pp. 221–231. Available at `http://dl.acm.org/citation.cfm?id=646239.683375`.

[11] Joseph A. Goguen & José Meseguer (1992): *Order-sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations*. Theor. Comput. Sci. 105(2), pp. 217–273, doi:10.1016/0304-3975(92)90302-V. Available at `http://dx.doi.org/10.1016/0304-3975(92)90302-V`.

[12] Chris Hathhorn, Chucky Ellison & Grigore Roşu (2015): *Defining the Undefinedness of C*. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15), ACM, pp. 336–345, doi:http://dx.doi.org/10.1145/2813885.2737979.

[13] Narciso Martí-Oliet & José Meseguer (2002): *Rewriting Logic as a Logical and Semantic Framework*, pp. 1–87. Springer Netherlands, Dordrecht, doi:10.1007/978-94-017-0464-9_1. Available at `http://dx.doi.org/10.1007/978-94-017-0464-9_1`.

[14] José Meseguer (2003): *Software specification and verification in rewriting logic*. NATO SCIENCE SERIES SUB SERIES III COMPUTER AND SYSTEMS SCIENCES 191, pp. 133–194.

[15] José Meseguer, Joseph A. Goguen & Gert Smolka (1989): *Order-sorted Unification*. J. Symb. Comput. 8(4), pp. 383–413, doi:10.1016/S0747-7171(89)80036-7. Available at `http://dx.doi.org/10.1016/S0747-7171(89)80036-7`.

[16] José Meseguer & Stephen Skeirik (2017): *Equational formulas and pattern operations in initial order-sorted algebras*. *Formal Aspects of Computing* 29(3), pp. 423–452, doi:10.1007/s00165-017-0415-5. Available at `http://dx.doi.org/10.1007/s00165-017-0415-5`.

[17] Robin Milner, Mads Tofte & David Macqueen (1997): *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

[18] Daejun Park, Andrei Ştefănescu & Grigore Roşu (2015): *KJS: A Complete Formal Semantics of JavaScript*. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, ACM, pp. 346–356, doi:http://dx.doi.org/10.1145/2737924.2737991.

[19] Lawrence C. Paulson (1990): *Isabelle: The Next 700 Theorem Provers*. In P. Odifreddi, editor: *Logic and Computer Science*, Academic Press, pp. 361–386.

[20] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the K Semantic Framework*. *Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012. Available at `http://dx.doi.org/10.1016/j.jlap.2010.03.012`.

[21] Hao Wang (1952): *Logic of many-sorted theories*. *Journal of Symbolic Logic* 17(2), p. 105âĂŞ116, doi:10.2307/2266241.