A Appendix

A.1 An Example of $\mathbb{K}$

An Example – IMP. We first briefly introduce an example $\mathbb{K}$ specification, IMP (Fig. 6), in the FAST form, to highlight the features of defining a language in $\mathbb{K}$. The figure contains the IMP specification with most of its syntax and some semantic definitions, and we assume in IMP that all program variables are heap ones that can be shared through different threads.

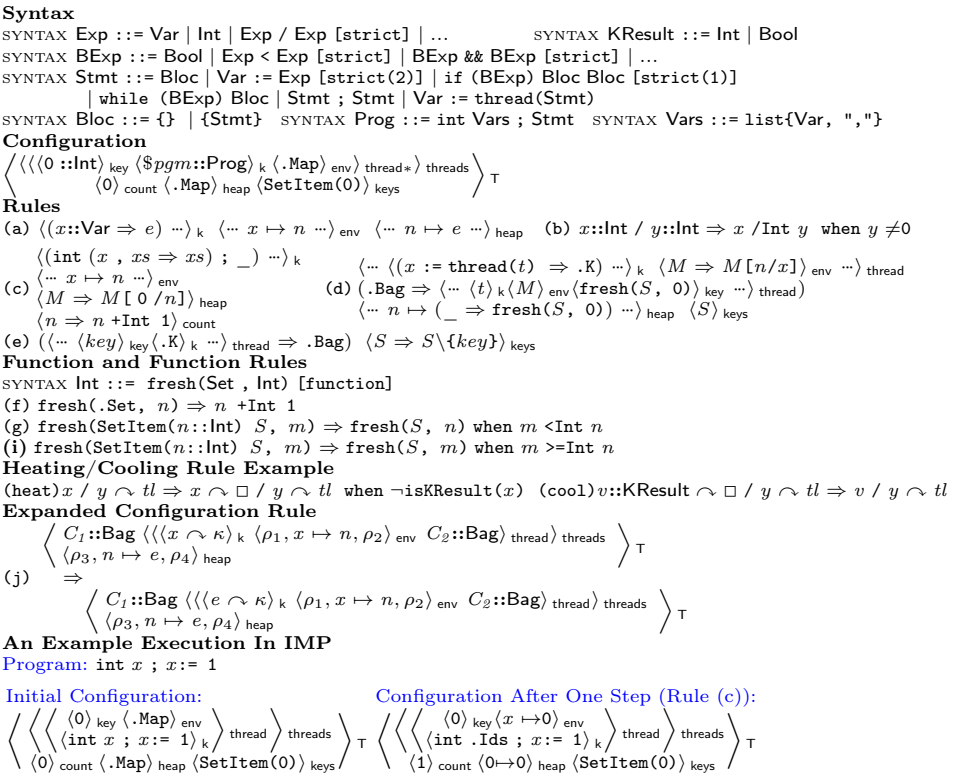

Fig. 6: A Summary of $\mathbb{K}$ by IMP

Syntax in a $\mathbb{K}$ Theory. In $\mathbb{K}$, the keyword SYNTAX introduces a finite set of syntactic definitions, separated by "|", such as the definition of the sort Exp. Each syntactic definition is a list of names. The names in Sans Serif font are non-terminals (sorts), while the names in TT font are terminals. A syntactic definition (e.g. Exp ::= Var) introducing only a singleton sort defines a relation that subsorts the singleton sort (Var) to the target sort (Exp). The definition that subsorts sorts to KResult defines the evaluation result sorts in a specification.

Other kinds of syntactic definitions introduce user defined terms that express rules and programs. Every "real" syntactic definition (not subsorting) creates a prefix AST format like *KLabel*`(`*KList*`)`, where the *KLabel* term acts as a constructor automatically generated by the terminals and the structure of the definition, and the *KList* term is the argument list generated from the non-terminals of the definition. The syntax definitions in a 𝕂 theory are compiled by the **IsaK** static semantics into a sort set, a symbol table, a subsort relation and several heating/cooling rules as inputs for the **IsaK** dynamic semantics (Sec. 3).

𝕂 **Attributes in Syntax in FAST are Equal to Semantic Rules in BAST.** 𝕂 allows users to define attributes in a syntactic definition (written in brackets e.g. `[strict]`, in Fig. 6), some of which have semantic meanings. For example, the `strict(2)` attribute (in the definition: `Var := Exp [strict(2)]`) is compiled to a BAST theory that generates a pair of heating/cooling rules for the second non-terminal position of the term created by the definition. The `[strict]` attribute without any numbers indicates there is a pair of heating/cooling rules generated for each non-terminal position in the definition. We show an example pair of heating/cooling rules for the first non-terminal position of the "`/`" operator in Fig. 6. "$\curvearrowright$" is a list concatenation operator for connecting the computation sequence in a `k` cell, while "$\square$" is a special builtin operation in 𝕂 representing the removal of a redex subterm from a term and the creation of a "hole" waiting to be filled. The semantics of any heating rule is to break down a term into subterms and let other semantic rules evaluate them, while the meaning of any cooling rule is to merge an evaluated subterm to the right position ($\square$) of the term.

𝕂 **Configurations.** Allowing users to define a global initial configuration for every 𝕂 theory is a key 𝕂 feature. The initial configuration of a specification is an algebraic structure of the program states, which are organized as nested, labeled cells, in XML formats that hold semantic information, including the program itself (prefixed by the $ operator in Fig. 6). While the order of cells in a configuration is irrelevant, the contextual relations between cells are relevant and must be preserved by rules defined by users and subsequently "completely filled" in the compilation step in 𝕂 according to the configuration. In a trace evaluation, each step of the computations should produce a result state (configuration) that "matches" the structure of the initial configuration, meaning that the names, sorts, and structural relations of the cells are preserved in the result configuration and initial configuration. Leaf cells represent pieces of the program state, like computation stacks or continuations (e.g., `k`), environments (e.g., `env`), heaps (e.g., `heap`), etc. The content of each cell in an initial configuration has dual roles: it establishes the initial value of the computation and also defines the sort of the cell content. For example, the `key` cell in the IMP configuration (Fig. 6) is defined as `0` and sort `Int`; during an evaluation, the cell's initial value is `0`, and in every state of the evaluation, its content has a sort that subsorts to `Int`. The last part of Fig. 6 provides an example program in IMP combined with its initial configuration. After evaluating the initial configuration by rule `(c)` in IMP, the

contents of several cells are updated, but the structure relations and sorts of the cells are preserved during the evaluation.

Semantic Rules in a $\mathbb{K}$ Theory and Their Compilation in BAST. Fig. 6 also contains a set of IMP rules. The simplest form of the rules, such as rule `(b)`, describe behaviors that can happen in the first element position in a k cell (representing the computation list in a thread), without mentioning any cells or the tail of the computation list of the k cell. A little more complicated form of rules, such as `(heat)` and `(cool)`, mention the tail of the computation list (connected by "$\curvearrowright$"). They describe behaviors that can happen in a k cell, especially the relationships among different positions in the computation list. In the BAST format, these two kinds of rules are compiled to the same form (K rules). The most complicated form of rules, such as rule `(a)`, are typical configuration rules in $\mathbb{K}$, and they describe interactions among different device/state components in a system. For example, rule `(a)` reads from a value in the main memory (the heap cell) for a variable in the k cell through a local stack (env). The "⋯" operator in these rules represents portions of cells that are irrelevant. This unconventional notation allows users to write concise semantic rules.

In this paper, we focus on the dynamic semantics of $\mathbb{K}$. All these unconventional configuration rules are assumed to be compiled to a standard form (BAST) by the **IsaK** static semantics [27], and the dynamic semantics definitions are based on the compiled format. For example, rule (a) is compiled to rule (j). To translate (a) to (j), we would need to add the cells T, thread, threads in rule `(j)`, variables C_1 and C_2, and their sorts (Bag), to indicate the irrelevant program state pieces. Computations in the k cell would be separated by "$\curvearrowright$", which is now observable in `(j)`. The κ and ρ_1, ρ_2, ρ_3, and ρ_4 would fill in the place corresponding to the "⋯" in rule `(a)`.

In $\mathbb{K}$, configuration rules are also powerful enough to manipulate language device resources. For example, rules `(d)` and `(e)` create or finish a thread by adding or deleting a thread cell. These are handled by rewriting an empty Bag place (`.Bag`) to a new cell thread (rule `(d)`), or by rewriting a thread cell to an empty place (rule `(e)`). In $\mathbb{K}$, this is allowed only if the specific cell in the initial configuration (e.g. the configuration in Fig. 6) is marked as "$*$".

$\mathbb{K}$ also allows users to write equational rules, named function rules. The format is like the `fresh` definition in Fig. 6. Its syntactic definition (`fresh`) is labeled by an attribute `function`, and then the rules whose left-hand top-most constructor is the same as the *KLabel*'s term syntactic definition are recognized by $\mathbb{K}$ to be the function rules under the function definition. The left-hand-side of a valid function rule has argument sorts that subsort to the argument sorts defined in the function definition, and the target sort of the right-hand-side subsorts to the target sort of the definition. All of these rules are compiled to BAST (described in Sec. 3.1) as the input for the **IsaK** dynamic semantics.

A.2 IsaK BAST Syntax and Sorts

Domains and Terms

$v \in \mathcal{CN}ame \triangleq \mathcal{BN}ame \cup \{\mathsf{k}\}$ Config Names

$c \in \mathcal{KL}abel \triangleq \mathcal{LN}ame \cup \{\texttt{klabel}, \texttt{isKResult}, \wedge, \neg, =\}$
$\setminus\{\texttt{lConstr}, \texttt{sConstr}, \texttt{mConstr}, \texttt{bConstr}\}$ KLabels (Constructors)

$k \in \mathcal{KI}tem \triangleq \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}s \mid \square\texttt{::}s$ KItem Terms

$k \in \mathcal{K} \triangleq \mathcal{KI}tem\ list$ Associative and Identitive KItem Sequences

$kl \in \mathcal{KL}ist \triangleq \mathcal{K}\ list$ Associative and Identitive K Sequences

$\mathcal{L}istItem \triangleq \texttt{lConstr}(\mathcal{K})$ Singleton List Terms

$l \in \mathcal{L}istItem' \triangleq \mathcal{L}istItem \mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{List}$ List Terms With Funs

$l \in \mathcal{L}ist \triangleq \mathcal{L}istItem'\ list$ Associative and Identitive List Terms

$\mathcal{S}etItem \triangleq \texttt{sConstr}(\mathcal{K})$ Singleton Set Terms

$S \in \mathcal{S}etItem' \triangleq \mathcal{S}etItem \mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{Set}$ Set Terms With Funs

$S \in \mathcal{S}et \triangleq \mathcal{S}etItem'\ list$ Idempotent Set Terms

$\mathcal{M}apItem \triangleq \texttt{mConstr}(\mathcal{K}, \mathcal{K})$ Singleton Map Terms

$M \in \mathcal{M}apItem' \triangleq \mathcal{M}ap \mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{Map}$ Map Terms With Funs

$M \in \mathcal{M}ap \triangleq \mathcal{M}apItem'\ list$ Idempotent, and Functional Map Terms

$\mathcal{B}agItem \triangleq \texttt{bConstr}(\mathcal{CN}ame, \mathcal{T}erm)$ Singleton Configurations

$C \in \mathcal{B}ag \triangleq \mathcal{B}agItem\ list$
Associative, Commutative, and Identitive Configuration Terms

$t \in \mathcal{T}erm \triangleq \mathcal{KI}tem \cup \mathcal{K} \cup \mathcal{L}ist \cup \mathcal{S}et \cup \mathcal{M}ap \cup \mathcal{B}ag$ Allowed Terms

$\mathcal{P}at \triangleq \mathcal{T}erm$ Patterns $\mathcal{E}xp \triangleq \mathcal{T}erm$ Expressions

Transition Rule Syntax

$rl \in \mathcal{R}ule \triangleq$
$\mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}s \Rightarrow \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}s_1\,\texttt{when}\ \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}s_2$
(* KItem Function Rules($s_1 \sqsubseteq s \sqsubseteq \mathsf{KItem} \wedge s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{K} \Rightarrow (\mathcal{K} \mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{K})\,\texttt{when}\ \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}s_2$
(* K Function Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{List} \Rightarrow (\mathcal{L}ist \mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{List})$
$\texttt{when}\ \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}s_2$ (* List Function Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{Set} \Rightarrow (\mathcal{S}et \mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{Set})$
$\texttt{when}\ \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}s_2$ (* Set Function Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{Map} \Rightarrow (\mathcal{M}ap \mid \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}\mathsf{Map})$
$\texttt{when}\ \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}s_2$ (* Map Function Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{K} \Rightarrow \mathcal{K}\ \texttt{when}\ \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}s_2$ (* K Transition Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{B}ag \Rightarrow \mathcal{B}ag\ \texttt{when}\ \mathcal{KL}abel\,(\mathcal{KL}ist)\texttt{::}s_2$
(* Configuration Transition Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)

IsaK Theory Input

$\mathcal{S}ymbol \triangleq \mathcal{KL}abel \mid \mathcal{KL}abel \rightarrow \mathcal{B}ool$ Symbols

$\Upsilon \subseteq \Psi \times \Psi\ list \times \mathcal{S}ymbol \times \mathcal{B}ool$ Symbol Table

Rule Set: $\Delta \subseteq \mathcal{R}ule$ Subsort Relation: $\sqsubseteq$

Fig. 7: **IsaK** Syntax in Isabelle (No Meta-Variables)

We first introduce the syntactic formulation of a given **IsaK** theory in BAST before we introduce the semantics of evaluating a program for a theory. Syntactically, every **IsaK** theory is expressed as a tuple of $(\Psi, \sqsubseteq, \Upsilon, \Delta)$, where Ψ is a set of sort names and $(\Psi, \sqsubseteq)$ is a poset, Υ is a symbol table and Δ is a set of $\mathcal{Rule}$ terms, which will be introduced later in the section. $\sqsubseteq$ is a subsort relation built on pairs of sorts in Ψ. We have restrictions on Ψ and $\sqsubseteq$ as follows:

Sorts
$\mathcal{SystemSort} \triangleq \{\mathsf{K}, \mathsf{KItem}, \mathsf{KList}, \mathsf{List}, \mathsf{Set}, \mathsf{Map}, \mathsf{Bag}\}$
$\mathcal{BuiltinSort} \triangleq \mathcal{SystemSort} \cup \{\mathsf{Bool}\}$
$\mathcal{RName} \subseteq \mathcal{UsrSort} \qquad \mathcal{BuiltinSort} \cap \mathcal{UsrSort} = \emptyset$
$\mathcal{ResultSort} \triangleq \mathcal{RName} \cup \{\mathsf{Bool}\}$
$s \in \Psi \triangleq \mathcal{UsrSort} \cup \mathcal{BuiltinSort}$
Sort/Subsort Sat Properties
$(\Psi, \sqsubseteq)$ is a poset
$\sqsubseteq \supseteq (\mathcal{UsrSort} \times \{\mathsf{KItem}\}) \cup \{(\mathsf{KItem}, \mathsf{K})\}$
$\forall s_1\ s_2.\ s_1 \in \{\mathsf{KList}, \mathsf{List}, \mathsf{Set}, \mathsf{Map}, \mathsf{Bag}\} \wedge s_1 \sqsubseteq s_2 \Rightarrow s_1 = s_2$

Every sort is disjointly either a user-defined sort ($\mathcal{UsrSort}$) or a built-in sort ($\mathcal{BuiltinSort}$). Each sort in $\mathcal{ResultSort}$ is either Bool, or a user-defined sort that can be the sort of the result of a computation, like Int (see Fig. 6). There are several restrictions on $\sqsubseteq$. For example, sort K is an upper bound of $\mathcal{UsrSort}$, while KItem is the supremum of the same set. The elements in {KList, List, Set, Map, Bag} are incomparable under $\sqsubseteq$; and $\mathcal{SystemSorts}$ are not result sorts.

Additionally, in the original $\mathbb{K}$, when a result sort is declared, the sort is subsorted to a special sort KResult. This formalization causes a problem in the type (sort) system soundness in $\mathbb{K}$: a term with a result sort can be rewritten to another result-sorted term, but the position holding the term is defined to hold one of the sorts but not the other one. For example, assume that x has value `true` in the heap, and we want to compute x`/1`. By applying rules `(heat)` and `(a)` in Fig. 6, the result is the term: `true` $\curvearrowright$ ($\square$`/1`). Since `true` is a KResult term, we can use rule `(cool)` in Fig. 6 to rewrite the term to `true/1`. This term is clearly ill-typed. In an evaluation in a $\mathbb{K}$ theory, this feature makes some rule applications result in type-errors that cannot make any further evaluations, but the $\mathbb{K}$ type (sort) system cannot detect this error in the theory. In **IsaK**, we discard the KResult sort and view the sorts subsorting to KResult as defining a set of result sorts. We use the predicate `isKResult`, whose meaning is the membership of $\mathcal{ResultSort}$. We replace every place in a $\mathbb{K}$ theory that describes a term subsorting to KResult with an `isKResult` predicate on the term. Thus, the subsort relation of KResult in a $\mathbb{K}$ theory is replaced in **IsaK** by the checking of a property on terms by `isKResult`. For example, the sort enforcement of KResult in rule `(cool)` becomes:

$$v{:}\mathsf{Exp} \curvearrowright \square \ /\ y \curvearrowright tl \Rightarrow v \ /\ y \curvearrowright tl \ \texttt{when isKResult}(v)$$

We now describe **IsaK** terms through the symbol table Υ. Any term in **IsaK** satisfies the grammar defined in Isabelle in Fig. 7. The symbol table (Υ) is a

translated product of the **IsaK** static semantics in Sec. 2, and each of its entries describes a syntactic definition for a specific constructor. This is represented as a tuple of a target sort (s), a list of argument sorts (sl), a set of symbol names (CS) representing a set of constructors that is either a singleton set of a *KLabel* term or set of many *KLabel* terms (generated from user defined tokens, like variable names and integers), and a Boolean value (b) indicating if the constructor is a function constructor. For a given **IsaK** theory, the constructors (having the type *KLabel*) appearing in the AST tree of a term must be a constructor name (*Symbol*) in an entry of Υ. For a given symbol table entry (s, sl, CS, b), the sort information for the constructor $c \in CS$ is $sl \to s$. A valid term in an **IsaK** theory satisfies the following definition.

Definition 1. Given a symbol table Υ, a term is a valid **IsaK** term iff every subterm (having the form $c(c_1, ..., c_n)$**::**s) appearing in the AST of the term satisfies the following:

- If $c_1, ..., c_n$ is an empty list, then s is a supersort of the sort of c in Υ.
- If $c_1, ..., c_n$ is not empty, let $s'_1, ..., s'_n$ be the argument sorts of c in Υ. Then, for every term c_i in $c_1, ..., c_n$, its sort s_i is a subsort of s'_i, and the sort s is a supersort of the sort of c in Υ.

$\mathbb{K}$ is a language that allows users to define a specification by giving a set of terms containing meta-variables, and it also allows them to define a specific ground term (without any meta-variable) as a "program" that produces a trace of the states when executed with the rules defined in the specification. An **IsaK** theory represents the specification, doing so as a tuple of $(\Psi, \sqsubseteq, \Upsilon, \Delta)$. Δ is a finite set of rules, each of which is a *Rule* term (possibly with meta-variables) defined in Fig. 7. A "program" for the theory is a *Bag* ground term.

In Fig. 7, we now describe briefly the grammars that define the Υ set in an **IsaK** theory. The variables appearing on the left (before $\in$) range over the sets on the right. We assume that the name sets (*UsrSort*, *LName*, and *BName*) are all disjointly unioned with each other. Any term in *KItem* is a user defined one allowed in a computation, with the fixed format of a constructor (*KLabel*) applied to a list of arguments (*KList*). The operation (**::**) represents a type enforcement by giving a sort. It can appear in any term in **IsaK**, and sometimes we omit such information in examples.

The $\square$ symbol in the *KItem* definition in Fig. 7 represents a family of symbols, one for each sort, each which represents a "hole" in the context term created when we split a term into context and redex terms, such as the $\square$ in Fig. 6 `(heat)`. These symbols are mainly used for $\mathbb{K}$ heating/cooling rules. With the $\square$ symbols, the evaluations of a term using heating/cooling rules are the same as for other rules. In **IsaK**, there are built-in lists (*List*), sets (*Set*), and maps (*Map*) for users with different built-in equational properties (listed in Fig. 7). Type *Bag* contains lists (with associative, commutative, and identity equational properties) of basic program state pieces, named configuration pieces or cells and having the type *BagItem*. Each cell contains a cell name (*CName*) and sub-configuration components, an example of which the IMP configuration is shown

in Fig. 6. In Sec. 2, we introduced how a configuration works. Users need to define an initial configuration along with their language specification. For every step of the evaluation of an input program, the result program state obeys the sort and position relations among the different cells in the initial configuration. One feature in **IsaK** that is useful for configuration translation (Sec. 4.2) is that Υ contains entries for all cell names appearing in the initial configuration. In an **IsaK** theory, the name of each cell has an entry in Υ that contains a target sort the same as the sort of the element in the cell in the initial configuration, an empty argument sort list, a singleton set of the cell name, and a `false` Boolean value. For example, cell `env` has a content of sort `Map` (Fig. 6), so it has the entry: $(\mathsf{Map}, [], \{\mathtt{env}\}, \mathtt{false})$.

Besides the above syntactic definitions and restrictions, an **IsaK** theory also has other syntactic restrictions that appear in the static translation process from FAST to BAST (introduced in [27]). For example, all rules ($\mathcal{R}ule$) have the format: $\mathcal{P}at \Rightarrow \mathcal{E}xp$ `when` $\mathcal{E}xp$, where the left hand side of $\Rightarrow$ is the pattern ($\mathcal{P}at$) to match with, and the right hand side of $\Rightarrow$ is the target expression ($\mathcal{E}xp$) to rewrite to, provided that the condition expression ($\mathcal{E}xp$) after the keyword `when` is satisfied. Terms in both types $\mathcal{P}at$ and $\mathcal{E}xp$ are **IsaK** terms ($\mathcal{T}erm$), but they have different syntactic restrictions checked by the **IsaK** static semantics (Supplement). For example, no term in $\mathcal{P}at$ can have proper sub-terms possessing function constructors. For a given $\mathcal{R}ule$ term, meta-variables can only represent a term ($\mathcal{T}erm$ffr in Fig. 7).

A.3 Translating Datatypes

For a given **IsaK** theory $\Theta = (\Psi, \sqsubseteq, \Upsilon, \Delta)$, we first translate the tuple $(\Psi, \sqsubseteq, \Upsilon)$ to a pair of a finite quotient type set and a finite set of Isabelle proofs (Ω^q, Π) in the translated Isabelle theory (Ξ), such that all relations in $\sqsubseteq$ are invisible in Ξ, but their functionality is merged in Ω^q. The way to achieve this is to utilize Isabelle quotient types: we first translate the **IsaK** datatype tuples $(\Psi, \sqsubseteq, \Upsilon)$ to a finite Isabelle datatype set Ω by explicitly coercing every pair in $\sqsubseteq$, and then translate Ω to a quotient type set Ω^q with a finite set of proofs (Π), one for each target sort in Ω^q, to show that each quotient type in Ω^q defines an equivalence relation over all of the syntax defined in Ω. We describe the two processes below.

The Translation from $\mathbb{K}$ Datatypes to Isabelle Datatypes. The translation step from the tuple $(\Psi, \sqsubseteq, \Upsilon)$ to an Isabelle datatype set Ω has two parts: adding builtin datatypes (corresponding to terms in $\mathcal{B}uiltinSort$ in Sec. 3.1) and translating user defined datatypes (corresponding to terms in $\mathcal{U}srSort$ in Sec. 3.1). The two parts for translated result of the **IsaK** theory (IMP) syntax in Fig. 2 are shown Fig. 8. The builtin datatypes that are additionally generated in Fig. 8 are in a one-to-one correspondence with the datatypes in **IsaK** in Fig. 7, except the datatypes $\mathcal{K}Label$ / $\mathcal{K}List$, which represent constructors and their arguments in $\mathbb{K}$ and are absorbed into different datatypes in Isabelle. We implement the builtin $\mathcal{K}$, $\mathcal{L}ist$, $\mathcal{S}et$, $\mathcal{M}ap$, and $\mathcal{B}ag$ datatypes as type synonyms for Isabelle builtin lists of corresponding singleton item datatypes, e.g. $\mathcal{K}Item\ list$ for $\mathcal{K}$. The reason to translate these builtin datatypes to Isabelle builtin list structures is to capture

Builtins
datatype *KItem* = s_1_KItem s_1 |...| s_n_KItem s_n $s_1, ..., s_n \in$ *UsrSort*
type_synonym *K* = *KItem list*
datatype *SetItem* = SConstr *K* type_synonym *Set* = *SetItem list*
datatype *ListItem* = LConstr *K* type_synonym *List* = *ListItem list*
datatype *BagItem* = BagC *CName Bag* | MapC *CName Map* | SetC *CName Set*
| ListC *CName List* | KC *CName K*
type_synonym *Bag* = *BagItem list* datatype *MapItem* = MConstr *K K*
type_synonym *Map* = *MapItem list*
Translated Syntax
datatype *Exp* = Exp_Hole | Var_Exp *Var* | Int_Exp *Int* | Div *Exp Exp* | ...
datatype *BExp* = BExp_Hole | Bool_Exp *Bool* | Less *Exp Exp* | And *Exp Exp* | ...
datatype *Stmt* = Stmt_Hole | Bloc_Stmt *Bloc* | Assign *Var Exp*
| If *BExp Bloc Bloc* | While *BExp Bloc* | Seq *Stmt Stmt* | Thread *Var Stmt*
datatype *Bloc* = Empty | Single *Stmt* datatype *Prog* = Prog *Vars Stmt*
datatype*Vars* = VarUnit | VarCons *Var Vars*

Fig. 8: Example of Datatype Translation (IMP)

the aspect that some builtin datatypes have implicit equational properties associated with them (listed in Fig. 7). By representing these datatypes as Isabelle list structures and representing a connection operation in **IsaK** (e.g. the set concatenation operation in $\mathbb{K}$) as an Isabelle list concatenation operation (@), we are able to capture the implicit associative and identity equational properties on the these datatypes without extra cares. The other implicit equational properties are dealt with when translating datatypes to quotient types.

The translation of user defined datatypes in $\mathbb{K}$ to Isabelle is done by adding explicit coercions for all subsort relation pairs in $\sqsubseteq$, e.g. the constructor `Var_Exp` coerces a term in *Var* to *Exp*, except that all function constructs (e.g. `fresh` in Fig. 6), which are translated directly into inductive relations without having datatype definitions in Isabelle (Sec. 4.2). Additionally, Since every user defined sort (s) is a subsort of KItem, we implement *KItem* as the union of all coercions of user defined sorts by adding a constructor for each one (s) of them as: `s_KItem`. We also add an extra constructor (like `Exp_Hole`) for each sort that contains some syntactic definitions with `[strict]` attributes to represent the $\square$ term in **IsaK** (Sec. 3). In IMP (Fig. 8), for example, we generate extra "hole" constructs for the types *Exp*, *BExp*, and *Stmt*, but other user defined sorts have no such construct because they do not have a definition with a `[strict]` attribute (Fig. 6).

From Datatypes to Quotient Types. Here we translate the Isabelle datatype set Ω to the quotient type set Ω^q with a set of proofs Π. A quotient type represents a set of terms, with a fixed target sort, whose elements are equivalence classes that are partitioned the whole term domain by a given set of equations. Some datatypes in Ω are only translated to "trivial" quotient types, meaning a quotient type with the Isabelle's builtin = operation as its equivalence relation.

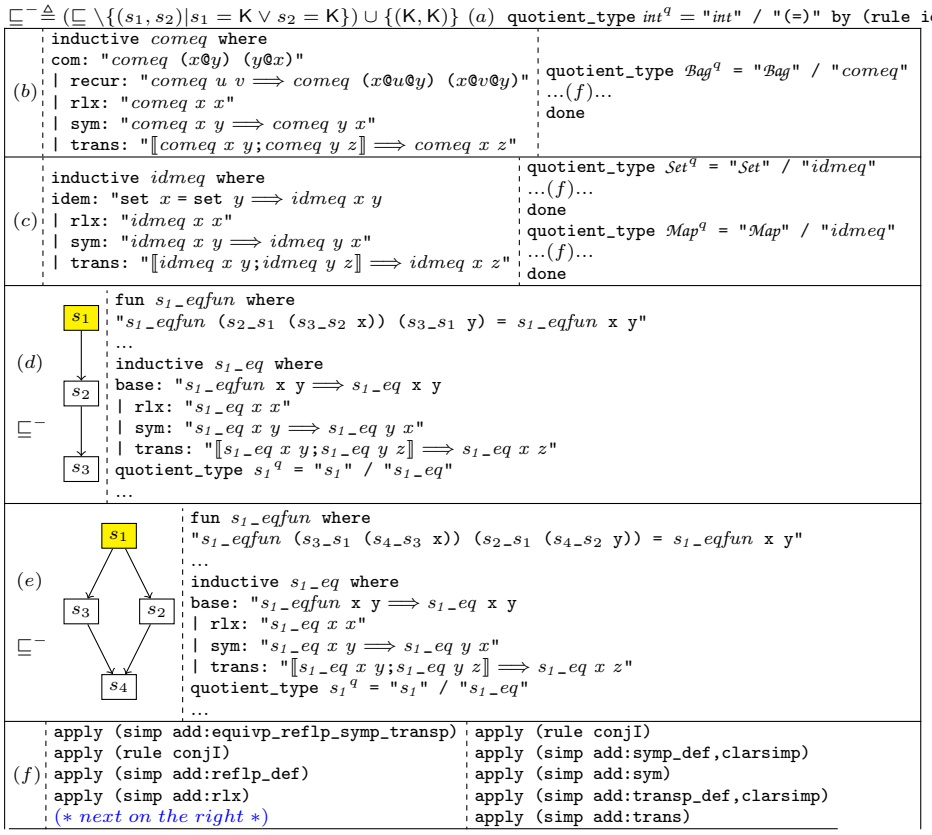

Fig. 9: Example of Translation to Quotient Types

The translation of *Int* to a quotient type in (*a*) (Fig. 8) is one example, and we just need the one-line proof "`rule identity_equivp`" for such a case.

For any datatype subset of Ω indexed by a specific target sort, there are four cases necessarily needing non-trivial quotient type translations. The general strategy for translating non-trivial cases is to define inductive relations to capture equivalence relations defined for the quotient types, and to prove that these really are such relations. Among these inductive equivalence relation definitions for translating non-trivial cases, we define the `rlx`, `sym`, and `trans` rules to ensure that the definitions are equivalence relations, such as the ones in (*b*) and (*c*) in Fig. 9, which capture the implicit equational properties (only the communicative and idempotent properties) hiding in the builtin terms *Bag*, *Map* and *Set*. Case (*b*) deals with the communicative equational property in *Bag* terms. The `com` and `recur` rules capture the communicative relations among the elements in a *BagItem list* (which is a *Bag* term) precisely. In the case, the right figure shows

how a quotient type with a proof is defined in Isabelle, and the proof content is in (f) in Fig. 9. In fact, (f) is the generalized proof for every non-trivial case quotient type proof in the translation. Case (c) provides an inductive relation capturing the idempotent equational property in the *Set* and *Map* terms. The rule `idem` defines the core equivalence property of two lists of *SetItem* or *MapItem* elements: two lists are equivalent if the set translations of the two lists are the same. As we stated in Fig. 7, *Map* terms must also be functional to be valid in a configuration. We incorporate the functional property as a transition rule in Sec. 4.2. Cases (d) and (e) in Fig. 9 capture all necessary non-trivial cases translating from datatypes to quotient types for user defined sorts/datatypes, whose general concept has been described in [25]. The process is to identify the possibly equivalent terms due to the removal of the subsort relation and explicit coercions when translating an order-sorted algebra to a many-sorted one. We first manipulate the input subsort relation $\sqsubseteq$ to be the definition of $\sqsubseteq^-$ in Fig. 9 by eliminating all subsorts related to sort K. Here is the reason. The only immediate subsort of sort K in $\sqsubseteq$ is the sort KItem, and users are not allowed to subsort other sorts to K. The only equivalent terms caused by explicit coercing KItem to K are those recognizing a KItem term as a singleton K term, which will be translated properly at the stage of translating terms and rules (Sec. 4.2). Cases (d) (and (e)) in Fig. 9 describe how we generate quotient types when a "line" (and a "diamond") structure is presented in the subsort relation $\sqsubseteq^-$. For the terms in the sort marked as yellow in cases (d) and (e), we generate a function, a relation, and a proof to capture capture the equivalence relations among these terms. Case (d) describes a possible "line" structure in $\sqsubseteq^-$, where three different sorts s_1, s_2, and s_3 have subsort relations in a line, like the left graph in (d). In this case, if the terms in s_1 have a combined explicit coercion $(s_2\_s_1\ (s_3\_s_2\ \texttt{x}))$, it is equivalent to a term being directly coerced from s_3, as $(s_3\_s_1\ \texttt{y})$, provided that the terms `x` and `y` are also equivalent. In Isabelle, we generate a function s_1*_eqfun* to capture the above description. The ... part contains other trivial cases for two terms in s_1. Sometimes, if a target sort s_1 contains other possible subsort relations fitting patterns in (d) and (e), we also need to take care of those situations in s_1*_eqfun*. The inductive relation s_1*_eq* is a trivial equivalence relation wrapper for s_1*_eqfun*. We can then build the quotient type ${s_1}^q$ based on s_1*_eq* with the same proof as (f). Case (e) describes a possible "diamond" structure in $\sqsubseteq^-$. In four different sorts s_1, s_2, s_3, and s_4, the sorts s_1, s_2, and s_4 have subsort relations, while the sorts s_1, s_3, and s_4 also have subsort relations. In this case, the coercions using different paths from s_4 to s_1 are all equivalent. We implement case (e) by the function s_1*_eqfun*, relation s_1*_eq* and quotient type s_1^q, in the same manner for case (d).

A.4 Translating $\mathbb{K}$ Terms and Rules

Here we translate the **IsaK** terms and rules. For an **IsaK** theory $(\Psi, \sqsubseteq, \Upsilon, \Delta)$, the translation algorithm for user defined terms simply walks down the ASTs of the terms by adding explicit coercions according to the syntactic translations in Sec. 4.1. The only tricky aspect is that the terms translated in Isabelle have no

KLabel or *KList* subterms. The translation of builtin terms can be summarized as the translation of **IsaK** configurations to Isabelle ones. The translation algorithm is straightforward with keeping an eye on the symbol table Υ to determine the sort for every cell in the configuration.

(*a*)
```
BagC T [BagC threads [BagC thread [KC key [Int_KItem 0],
    KC k [Prog (VarCons x VarUnit) (Assign x (Int_Exp 1))],MapC env []]], KC count [Int_KItem 1],
    MapC heap [MConstr [Int_KItem 0] [Int_KItem 0]], SetC keys [SConstr [Int_KItem 0]]]
```

(*b*)
```
c(kl)::s ⇒ c1(kl1)::s1 when c2(kl2)::s2
```
```
φ1 ∧ ... ∧ φn ⟹ c_ind t1 t2
```

```
fresh(SetItem(n::Int) S, m)
   ⇒ fresh(S, n) when m <Int n
...
```
```
inductive fresh_ind where
⟦m < n; fresh_ind (S, n) x1⟧
⟹ fresh_ind ([SConstr [Int_KItem n]@S, m) x1
...
definition fresh where
"fresh e = (SOME x . fresh_ind e x)"
```

(*c*)
```
t1 ⇒ t2 when c(kl)::s
```
```
φ1 ∧ ... ∧ φn ⟹ τ_rule t3 t4
```

```
v:Exp ↷ □ / y ↷ tl ⇒ v / y ↷ tl when isKResult(v)
```
```
inductive k_rule where
...
⟦t = abs_K((Exp_KItem v)#((Div x Exp_Hole)#tl));
isKResult((Exp_KItems v)); t' = abs_K((Div x v)#tl)⟧
    ⟹ k_rule t t'
...
```

(*d*)
```
inductive bag_rule where
  ⟦locate (abs_Bag C) = (C', t); k_rule (abs_K t) t'⟧ ⟹ bag_rule C (abs_Bag C'[rep_K t'])
| ⟦C = abs_Bag (BagC T ([BagC threads [BagC thread [KC key [key],KC k []]@xs]@ys, SetC keys S]@zs));
  C' = abs_Bag (BagC T ([BagC threads ys, SetC keys (SetCut(S, [SetConstr [Int_KItem key]]))]@zs))⟧
    ⟹ bag_rule C C'
  ...
```

(*e*)
```
inductive top where
    ⟦is_map_fun C1; C = abs_Bag C1; bag_rule C C' ⟧ ⟹ top C (Some C')
  | ⟦¬is_map_fun C1; C = abs_Bag C1⟧ ⟹ top C None
```

Fig. 10: Examples of the Translation of Terms and Rules

Fig. 10 (*a*) is a translated term from the initial configuration in Fig. 6; its datatype definition is in Fig. 8. To determine the constructor for cell T (BagC or MapC, etc), we look at Υ for the target sort of T. Since it has sort Bag, we add the constructor BagC for cell T. If the target sort of a cell (e.g. key) subsorts to K, we give the cell a constructor KC, and turn the cell content to a singleton sort K term, such as the translated term [Int_KItem 0] in the cell key. One of the benefits of using **TransK** instead of **IsaK** in Isabelle is its significantly shorter representations of terms. In fact, the initial configuration in (*a*) (Fig. 10) is one-third the length of what it would be if written in **IsaK**.

Next, we introduce the translation of rules, which is to translate the rule set Δ to a set of rules Δ^i, whose elements are all represented as inductive relations in Isabelle. The translated relations are all quantifier-free with all meta-variables represented as universally quantified meta-variables in Isabelle. In Sec. 3, we introduced the **IsaK** rewriting system by dividing rules into three kinds: function, K, and configuration rules. The rule translation deals with these rules differently. The functional checking step in the common evaluation procedures (Sec. 3.2) is disregarded from the rule translation here and will be represented as a specific

inductive rule to check that every *Map* term in a configuration is functional in the latter part of the section.

Translating Function Rules. We first investigate the translation of function rules. Each rule translation is divided into two parts: a translated inductive relation in Isabelle that captures the meaning of the function rule, and a definition using Hilbert's choice operator to produce the output of the relation. In $\mathbb{K}$, a function is defined as a syntactic definition with several function rewrite rules, whose format is as (b) in Fig. 10. Each function is translated to a single inductive relation and a definition using Hilbert's choice operator. Given a subset of the symbol table Υ (as Υ_f) containing only function constructs, and a subset of Δ (as Δ_f) containing only function rules, we produce a set of inductive relations in Isabelle as Δ_f^i containing the translated results of Δ_f. In $\mathbb{K}$, applying a function rule rl_f on a given term t results in two possibilities: either it terminates and returns the resultant term t', or it never terminates due to endlessly rewriting the condition expression of rl_f. Moreover, in all $\mathbb{K}$ tools, function rule applications are implemented as a transition step in a big-step semantic format, where the function application step produces either an infinite sequence of function rule applications or the result of a finite sequence of function rules being applied to the input term. In **TransK**, we keep this strategy and translate function rules as inductive relations using a big-step semantic format.

Fig. 10 (b) describes the translation of a function rule to an Isabelle inductive relation with an example (based on the `fresh` function in Fig. 6). For each function label c, we select all function rules belonging to it in Δ_f (having the rule pattern of the top-most constructor being c). We generate an inductive relation header (c`_ind`) in Isabelle (e.g. the `fresh_ind` header in Fig. 10 (b)) for the group of rules belonging to c. For a single rule rl_c for the function label c, its translation results in an inductive relation case in the relation c`_ind`, where the term t_1 is the translated term describing the input pattern arguments of rl_c (the kl part). An example of such a pattern is the `(SetItem(`n`:Int) `S`, `m`)` part of the `fresh` function rule; it is translated to `([SConstr [Int_KItem `n`]@`S`, `m`)` in (b). The term t_2 is the translation of the target expression of rl_c. Sometimes we need to call rl_c or other functions recursively, so we might need to use a generated variable (x_1 in Fig. 10), and generate an equality in the condition of the inductive rule. The conditions $\varphi_1, ..., \varphi_n$ contains not only the translation of the condition expression of rl_c (the m `<Int` n part), but also the equities to access the recursive or other function calls. The handling of x_1 above is one example. If the rule expression (the t_2 and c_1`(`kl_1`)::`s_1 parts in (b)) contains other mutually recursive function calls, they also need to be translated into variable terms in t_2, with equities as some conditions in $\varphi_1, ..., \varphi_n$. After we construct the inductive relation for a $\mathbb{K}$ function (or inductive relations for a set of mutually recursive functions), we create a definition with the Hilbert's choice operator `SOME` to force the inductive relation to output terms with the type matching the target sort of the $\mathbb{K}$ function as the Isabelle definition in (b), so that we can use the name of the function in a configuration or other rule expressions directly.

Translating $\mathbb{K}$ and Configuration Rules. The general strategies for translating a K rule or a configuration rule are very similar, as described in Fig. 10 (c). They are almost the same as translating a function rule, except that t_3 and t_4 are mostly translated from the terms t_1 and t_2 that appear in the $\mathbb{K}$ or configuration rule. The τ in (c) is either `k` or `bag`. In (c), we show an example of translating the cooling rule example in Fig. 6 (actually, the revised of the example in Sec. 3.1) to an inductive relation case in the inductive relation `k_rule`. In the translation, the translated terms t and t' are quotient type terms of sort $\mathcal{K}$. This is why we use t and t' as variables and then place equities in the conditions in the case to enforce terms t and t' to be quotient typed terms.

Translating configuration rules is similar to translating K rules, except that configuration rules are applied to the whole program state (configuration). We translate all configuration rules in Δ to cases in an inductive relation named `bag_rule`, with an additional rule case capturing the applications of K rules. Fig. 10 (d) shows example rule cases in `bag_rule`. Given an input configuration C, the first case is to apply a K rule to a k cell in C. We pre-define a function `locate` in the case, along with the translated Isabelle theory, to locate a k cell in C and split C into a context $C[]_{\mathsf{k}}$ and a redex t (the content of the k cell). Then we apply the corresponding inductive relation `k_rule` to the quotient type term of t: (`abs_K` t), and merge the context and new redex as $C[\texttt{rep\_K}\ t']_{\mathsf{k}}$ as the new configuration. `abs_K` and `rep_K` are Isabelle functions to get an actual term from a quotient type term in sort K and vice versa. The second rule case in (d) is a translation from the `(j)` example rule in Fig. 6. Similar to the situation in translating K rules, the input and output configurations C and C' are quotient types, and their contents are wrapped in the coercion `abs_Bag`. Case (e) in Fig. 10 is the `top` inductive relation for a translated Isabelle theory. It implements the functional checking for every *Map* term in an input configuration C by a pre-defined function `is_map_fun`; and if the check is valid, then the `bag_rule` relation is allowed to apply to C and to observe one step transition; otherwise, the system enters an error state (`None`).

A.5 Type Theorem Proof Sketches based on the Isabelle Proof

In this section, we show proofs for Theorem 1 that is restated below as Theorem 4.

Theorem 4. For a type correct theory (Θ) without anywhere and macro rules, for any type correct configuration C, evaluating C in **IsaK** never results in a type-error.

Proof. Given an **IsaK** theory (Θ) and an initial configuration C that are type-checked, the proof is based on structure induction on different rules applied to configuration C. There are three possible rule applications:

- If the rule application is a function rule one, for any given configuration C that is split into a context $C[]^s_f$ and redex $c(kl)$**::**s, a function rule application

transit the term $c(kl)$**::**s into a possible new term t with sort s', and s' must subsort to the sort s based on the function rule type correctness (the target sort of the right-hand-side of a rule must be a subsort of the target sort of the left-hand-side). Thus, if we plug the term t back into the context $C[]^s_f$, the configuration becomes $C[t$**::**$s']^s_f$. Since s' subsorts to s, the new configuration $C[t$**::**$s']^s_f$ is type-correct.
- If the rule application is a K rule, for any given configuration C that is split into a context $C[]_{\mathsf{k}}$ and redex k that is a $\mathcal{K}$ term, a K rule only rewrites the term k to a new term k' with the same type $\mathcal{K}$. All $\mathcal{K}$ terms have sort K. Thus, the final result configuration $C[k']_{\mathsf{k}}$ is type-correct.
- If the rule application is a configuration rule, for any given configuration C, applying a configuration rule rewrites the term C to a new configuration C' that is type-correct.

Thus, evaluating a configuration C in the type-checked theory Θ never results in a type error.

A.6 IsaK and TransK Bisimulation Proof Sketches based on the Isabelle Proofs

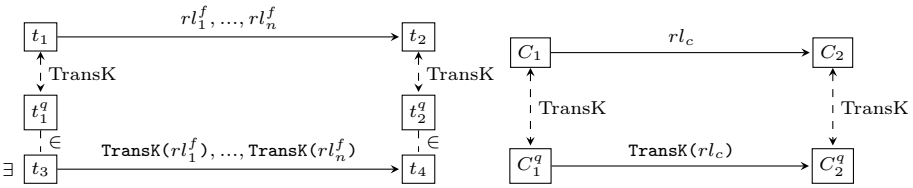

Fig. 11: Soundness and Completeness of **IsaK** and **TransK**

Here we construct the relationship between **IsaK** and **TransK**. Fig. 11 describes the general soundness and completeness proof diagrams between them. We have put a proof sketch in Appx. A.5 and the Isabelle formalization is at `https://github.com/liyili2/KtoIsabelle`.

We first look at the underlying system the proofs are based on. **IsaK** is defined in Isabelle, we also implemented **TransK** in Isabelle as well as a small Isabelle system (**Isab**) in Isabelle so that we could capture the semantics of the Isabelle theory translated from an **IsaK** one. If we assume that the generated quotient type proofs are always valid in Isabelle, which is obvious, an Isabelle theory translated by **TransK** only requires the soundness and completeness proofs involving Isabelle datatypes/quotient types and inductive relations. The translated inductive relations have the form as Fig. 10 (d), where every relation is a binary one rewriting from a term t_3 to a term t_4 with a list of conditions

$\varphi_1, ..., \varphi_n$, such that they are all quantifier-free. In addition, the definitions of functions require the support of Hilbert's choice operator. Thus, the rewriting semantics of the Isabelle system (**Isab**) supporting these features is just a simple typed λ-μ calculus with Hilbert's choice operator and quotient types. **Isab** is based on the λ-μ calculus developed by Matache et al. [31], and extended to support the rewriting of the inductive relations described above and definitions using Hilbert's choice operator.

To prove the soundness/completeness between an IsaK theory and its **TransK** translation into **Isab**, we have to prove the soundness and completeness of the functions rules separately from the K/configuration rules (Fig. 11). The problem is that every rule in **IsaK** has a conditional expression (having type *Bool*), and the rewrites of the *Bool* term can be infinite. Additionally, the function rules are translated to a definition of inductive relations in the big-step format, and it can be infifnite, too. The soundness and completeness for function rules have to assume that every rewrite of the function rule application on the conditional expression terminates in a finite sequence whose length is n. In addition, a function rule application in Isabelle deals with terms that are the translated datatypes not quotient types. Thus, a function rule is applied to a representative term in a given equivalence class, which is transitioned to another term as a representative in the resulting equivalence class, as described by the existential operation in the first diagram of Fig. 11. The term t_3 is a representative of the class t_1^q which is translated from the $\mathbb{K}$ term t_1. The soundness and completeness of function rule applications are described below.

Before we show Theorem 2 that is restated as Theorem 5. We also need to have the information in Fig. 9 that is restated as Fig. 9 here, as well as the information in Fig. 10 that is restated as Fig. 10 below.

Theorem 5. (Soundness) In **IsaK**, assume that a sequence of function rules $rl_1^f, ..., rl_n^f$ applied to a term t_1 terminates in n steps and results in term t_2, and t_1^q and t_2^q are quotient type terms in **Isab** translated by **TransK**, there exists a term t_3 in t_1^q transitioning through sequence of corresponding rule applications $\texttt{TransK}(rl_1^f), ..., \texttt{TransK}(rl_n^f)$ to term t_4, such that t_4 is in the quotient type class t_2^q, which is a translation from t_2.

(Completeness) If there exist quotient type terms t_1^q and t_2^q, such that a representative t_3 of t_1^q is transitioned to t_4 in t_2^q through a sequence of function rule applications $rl_1^{f'}, ..., rl_n^{f'}$, and $t_1^q = \texttt{TransK}(t_1)$, $t_2^q = \texttt{TransK}(t_2)$, and $rl_1^{f'} = \texttt{TransK}(rl_1^f)$,..., $rl_n^{f'} = \texttt{TransK}(rl_n^f)$, then t_1 is transitioned to t_2 through the sequence of function rule applications $rl_1^f, ..., rl_n^f$.

We first need to show the following lemma:

Lemma 1. For any two transformed terms t and t' in Isabelle, for any k and k', such that $t = \texttt{TransK}(k)$ and $t' = \texttt{TransK}(k')$, k and k' in **IsaK** are equivalent under the equational properties described in Fig. 7.

The proof of the lemma is based on an important assumption about $\mathbb{K}$/**IsaK**. One cannot define two terms in $\mathbb{K}$/**IsaK** with the same constructor. Even if some-

one makes two identical syntactic definitions in $\mathbb{K}$, the current $\mathbb{K}$ implementation helps map these two definitions with two distinct constructors. A consequence of this assumption is that every term has a least sort in $\mathbb{K}$/**IsaK**.

Proof. The proof is based on induction on the depth of the AST tree of a term t. The base step is simple, and we ignore it here. The inductive step is divisible into three cases:

- If the two terms t and t' are the same term, then the original terms k and k' are also the same.
- If the two terms t and t' are quotient-equivalent involving only *Set* and *Map* (like case (c) in Fig. 9), then k and k' are equivalent under the *Set* and *Map* equational properties listed in Fig. 7.
- If the two terms t and t' are quotient-equivalent through cases (d) and (e) in Fig. 9 (the line and diamond structures), then k and k' are the same term. This is because the constructors serving explicit coercions are created in the `TransK` function. The k and k' terms involve only subsort relations, without needing coercions. Once the coercions are removed, the terms k and k' are the same and have the same least sort, since every constructor in $\mathbb{K}$/**IsaK** is unique and has a unique least sort.

Next, we need to change the semantic rule in Fig. 4 a little. Specifically, we need to change case `(1)` to by adding a number n representing the number of steps an application of the rule can take. We need the step number to avoid the termination proof of a given **IsaK** theory.

$$\texttt{(1-a)}\ \frac{}{\texttt{true} \longrightarrow^{\Theta}_{0,f} \texttt{true}} \qquad \texttt{(1-b)}\ \frac{}{\texttt{false} \longrightarrow^{\Theta}_{0,f} \texttt{false}}$$

$$\texttt{(1-c)}\ \frac{c(kl) \neq \texttt{true} \qquad c(kl) \neq \texttt{false}}{c(kl) \longrightarrow^{\Theta}_{0,f} \texttt{error}}$$

$$\texttt{(1-d)}\ \frac{\begin{array}{c} rl \in \Theta \qquad m = \texttt{match}(rl, c(kl)) \qquad n \neq 0 \\ t = \texttt{subs}(m, \texttt{cond}(rl)) \qquad t\ (\longrightarrow^{\Theta}_{(n-1),f})^*\ \texttt{true} \end{array}}{c(kl) \longrightarrow^{\Theta}_{n,f} \texttt{subs}(m, \texttt{right}(rl))}$$

With the above rule change and Lemma 1, we can now prove Theorem 5.

Proof. We first show its soundness. The proof is rephrased as "for any number n, if applying a rule rl to a term t terminates in n steps, and produces a result of `true` or `false`, then applying $\texttt{TransK}(rl)$ to the term $\texttt{TransK}(t)$ also terminates in n step with a result of `True` or `False` in Isabelle."

We abbreviate the transition that happens in Isabelle as $\longrightarrow^{tr,\Theta}_{n,f}$.

We induct on the number n. The base step is when the term t is either `true` or `false` (otherwise, it is an `error`). Trivially, when translating the two terms in Isabelle, they are `True` or `False` values in Isabelle.

In the inductive step, for the n+1 step application, we have the assumption that the rule-application case `(1-d)` above is a valid one for term t. It means that

a mapping m results from $\texttt{match}(rl, t)$. We also have the translation of the rule $\texttt{TransK}(rl)$ and the term $\texttt{TransK}(t)$. By a case analysis of the different quotient translation cases in Fig. 9, we can conclude that there is a term t' which is in the same equivalence class as $\texttt{TransK}(t)$, such that we can find a mapping m' for $\texttt{match}(\texttt{TransK}(rl), \texttt{TransK}(t))$; and every mapped term t_i of meta-variable x_i in the mapping m' is in the same equivalence class as the term $\texttt{TransK}(m(x_i))$. In addition, translating the condition expression to `true` (through the transitions $(\longrightarrow^{\Theta}_{(n+1\ -1),f})^*$) is valid because of the inductive hypothesis ($n+1\ -1\ \leq n$). Thus, applying $\texttt{TransK}(rl)$ to the term $\texttt{TransK}(t)$ produces a valid term $\overline{t'}$ that is in the equivalence class of t', such that $t \longrightarrow^{\Theta}_{(n+1),f} t'$ via the rule rl.

We then show its completeness. The proof is rephrased as "for any number n, if we have a rule rl and a term t, and their translations $\texttt{TransK}(rl)$ and $\texttt{TransK}(t)$, then if an application of $\texttt{TransK}(rl)$ on the term $\texttt{TransK}(t)$ terminates in n steps with a result of `True` or `False` in Isabelle, then applying a rule rl to a term t terminates in n steps, and produces a result of `true` or `false`."

We also induct on the number n. The base step is when the term $\texttt{TransK}(t)$ is either `True` or `False` in Isabelle (otherwise, it is an `error`). Trivially, there is a term `true` or `false` in **IsaK** to match with the result term.

In the inductive step, for the $n+1$ step application, we have the assumption that the rule application case `(1-d)` above is a valid one for the term $\texttt{TransK}(t)$, and the rule being applied is $\texttt{TransK}(rl)$. This means that there is a mapping m as a result of $\texttt{match}(\texttt{TransK}(rl), \texttt{TransK}(t))$. We need to show that there is also a mapping m' as a result of $\texttt{match}(rl, t)$. We show that there is a term t' which is in the same equivalence class as t in **IsaK** by Lemma 1. By the lemma, there is a mapping $\texttt{match}(rl, t')$, such that every entry of m is the `TransK` translation result of the entry with the same meta-variable in m'. Furthermore, the terms t and t' are the same term because of the term normalization in **IsaK** introduced in Section 3.2. Every input term for a rule application is assumed to be term-normalized. If two terms are in the same equivalence class, the normalized term for all terms in the equivalence class is the same and it represents the canonical form of the equivalence class. Thus, there is a step transition $t \longrightarrow^{\Theta}_{(n+1),f} t''$ via rule rl, and the translation of term t'' is in the same equivalence class as the term $\overline{t''}$ in the transition $\texttt{TransK}(t) \longrightarrow^{tr,\Theta}_{(n+1),f} \overline{t''}$ via rule $\texttt{match}(\texttt{TransK}(rl))$.

We now show Theorem 3 that is restated as Theorem 6. We have a version of the theorem based on the revised Theorem 5 above, where every Boolean term is evaluated to a result of either `true` or `false` in n steps. Based on the revised "termination in n steps" Theorem 5.

Theorem 6. (Soundness) In **IsaK**, assume that a configuration C_1 is transitioned to C_2 through a $\mathbb{K}$ (or configuration) rule rl_c, and C_1^q is a quotient type term translated from C_1, then C_2^q is translated from C_2 by rule $\texttt{TransK}(rl_c)$.

(Completeness) If there exist quotient type configurations C_1^q and C_2^q, such that C_1^q transitions to C_2^q through a rule rl'_c, $C_1^q = \texttt{TransK}(C_1)$, and $rl'_c = \texttt{TransK}(rl_c)$, then C_2 is transitioned from C_1 by rule rl_c and $C_2^q = \texttt{TransK}(C_2)$.

We have the proof of Theorem 6 as follows:

Proof. We first show its soundness. Given an **IsaK** theory Θ and configuration C, we do a structural induction on different kinds of rules in Θ.

- If the application rule rl is a function rule, the configuration is split into a context $C[]^s_f$ and a redex $t{::}s$, such that $t \longrightarrow^{\Theta}_f t'$, and the final configuration is $C[t']^s_f$. The final configuration is well-typed because of Theorem 4. We then need to show that a translation of rl as $\texttt{TransK}(rl)$ can also be applied to term $\texttt{TransK}(t)$, and it is transitioned to a term $\overline{t'}$, which is in the same equivalence class of the translation result of t'. This step of the proof is similar to the one in proving Theorem 5, with the assumption that the rewrites of the condition expression of the rl rule with term t produces a Boolean result of `true` or `false` in n steps. We then need to show that the final configuration from applying function rule $\texttt{TransK}(rl)$ is in the same equivalence class as the term $\texttt{TransK}(C[t']^s_f)$. To accomplish this, we have a lemma stating that the application of an Isabelle `definition` to a term $\texttt{TransK}(C)$ (as in the case (b) translation result in Fig. 10) produces a term with the context $C'[]^s_f$ and redex $\overline{t'}{::}s$, where t' and the hole in $C'[]^s_f$ also have the type s, $\overline{t'}$ is in the same equivalence class as $\texttt{TransK}(t')$, and $C'[\overline{t'}]^s_f$ is in the same equivalence class as the term $\texttt{TransK}(C[t']^s_f)$.
- If the application rule rl is a K rule, the configuration is split into a context $C[]_{\mathsf{k}}$ and a redex k having sort K, such that applying rl to k results in k', and the final configuration is $C[k']_{\mathsf{k}}$. The final configuration is well-typed because of Theorem 4. We then need to show that the translation of rl as $\texttt{TransK}(rl)$ can be applied to the term $\texttt{TransK}(k)$ (like the `k_rule` translation in Fig. 10 (c)), and transition the term to a term $\overline{k'}$ that is in the equivalence class of term $\texttt{TransK}(k')$. This step of the proof is basically the same as in Theorem 5. Finally, we need to show that the transition rule in Isabelle (like the first line of Fig. 10 (d)) can split the configuration $\texttt{TransK}(C)$ into a context $C'[]_{\mathsf{k}}$ and redex $\overline{k}$, such that $\overline{k}$ is in the same equivalence class as $\texttt{TransK}(k)$, and $C'[k]_{\mathsf{k}}$ is in the same equivalence class as $\texttt{TransK}(C)$; and we need to show that the application of $\texttt{TransK}(rl)$ to $\overline{k}$ results in $\overline{k'}$, and $C'[\overline{k'}]_{\mathsf{k}}$ is in the same equivalence class as $\texttt{TransK}(C[k']_{\mathsf{k}})$. The proof of this step is done by a structural induction on any Isabelle term with the application of a special inductive rule like the one in the first line of Fig. 10 (d).
- If the application rule rl is a configuration rule, then we only need to show that when the configuration is transitioned to C' via the rule, the translation of C to $\texttt{TransK}(C)$ is also transitioned to $\overline{C'}$ via the rule $\texttt{TransK}(rl)$, such that $\overline{C'}$ is in the same equivalence class as $\texttt{TransK}(C')$. This is done by the same strategy as the proof of Theorem 5.

We then show its completeness. The proof is to do a rule induction on the `top` inductive relation listed in Fig. 10 (e), and then establish a sub-theorem by doing an induction on the rules in Fig. 10 (d). There are three important cases here:

- If the rule being applied is not the first line of Fig. 10 (d), then the rule is the translation of a configuration rule rl in the given **IsaK** theory Θ. By a

similar proof strategy as that for proving Theorem 5, we can then show that for any configuration $\overline{C} = \texttt{TransK}(C)$, the application of $\texttt{TransK}(rl)$ results in a new configuration $\overline{C'}$; the configuration C is also transitioned to a new configuration C' via rule rl; and $\texttt{TransK}(C')$ is in the same equivalence class as $\overline{C'}$.

- If the rule being applied is the first line of Fig. 10 (d), then the rule is the translation of K rule rl in the given **IsaK** theory Θ. Then, the given configuration $\overline{C} = \texttt{TransK}(C)$ is split into a context $\overline{C}[]_{\mathsf{k}}$ and redex $\overline{k}$. The rule application of $\texttt{TransK}(rl)$ to $\overline{k}$ results in a term $\overline{k'}$. We need to show that if $\overline{k} = \texttt{TransK}(k)$, then k transitions to k' via the K term transition part of rule rl, such that $\overline{k'}$ is in the same equivalence class as term $\texttt{TransK}(k')$. The proof of the existence of the term uses the same strategy as the one in the completeness proof of Theorem 5. Finally, we need to show that C is transitioned to $C[k']_{\mathsf{k}}$ via rule rl, such that $\overline{C}[\overline{k'}]_{\mathsf{k}}$ is in the same equivalence class as $C[k']_{\mathsf{k}}$. This proof is done by a similar strategy in proving the existence of the context-redex combination in the soundness proof of an K rule above.
- If a given configuration $\overline{C}$ contains an Isabelle definition t (like Fig. 10 (b)), then the definition must be translated from a function constructor appearing in the **IsaK** theory Θ. In this case, configuration $\overline{C}$ is moved through several transitions to a final step $\overline{C'}$ such that the definition is replaced by the ground term t' without any Isabelle definitions. In this case, $\overline{C}$ can be viewed as the split of a context $\overline{C}[]$ and a redex t, such that $\overline{C}[t] = \overline{C}$ and $\overline{C}[t'] = \overline{C'}$. We have shown inductively the following. For n steps of applications of the inductive rule associated with the definition $\overline{t}$, it is continuously translated to $\overline{t_1}, ..., \overline{t_n}$. For any rewrite from $\overline{t_i}$ to $\overline{t_{i+1}}$, the transition is through the inductive rule $\texttt{TransK}(rl)$, which is translated from a function rule rl in Θ. Let $\overline{t_i} = \texttt{TransK}(t_i)$, then t_i transitions to t_{i+1} via the application of rule rl, and $\texttt{TransK}(t_{i+1})$ is in the same equivalence class as $\overline{t_{i+1}}$. Thus, for any n step computations of inductive rule applications for an Isabelle definition, there are n step computations of function rule applications, and the final term t_n computed in the **IsaK** system is translated into the same equivalence class as the final result term $\overline{t_n}$ computed in the Isabelle system. Finally, let $\overline{t'} = \overline{t_n}$. We need to show that if $\overline{C} = \texttt{TransK}(C)$, $\overline{t} = \texttt{TransK}(t)$, and $\overline{t'} = \overline{t_n} = \texttt{TransK}(t_n)$, then $\texttt{TransK}(C[t_n])$ is in the same equivalence class as $\overline{C}[t']$. The proof of this step is the same process as we have already performed multiple times.

The above is the proof sketches of the Theorems 4, 5, and 6.