Viorel Preoteasa | Ralph-Johan Back | Johannes Eriksson

Verification and Code Generation for Invariant Diagrams in Isabelle

Turku Centre for Computer Science

TUCS Technical Report
No 1058, September 2012

Verification and Code Generation for Invariant Diagrams in Isabelle

Viorel Preoteasa
Ralph-Johan Back
Johannes Eriksson
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
{vpreotea,backrj,joheriks}@abo.fi

TUCS Technical Report
No 1058, September 2012

Abstract

Invariant-based programming is a correct-by-construction programming methodology in which programs are expressed as graphs of *situations* connected by *transitions*. Such graphs are called *invariant diagrams.* The situations correspond to the pre- and postconditions and loop invariants of the program, while the transitions correspond to the program statements. The situations are developed before the transitions, and each transition is verified at the time it is added to the diagram. The correctness conditions for the transitions are derived using Hoare-like rules. In this paper, we present an embedding of invariant diagrams in the higher-order logic framework Isabelle/HOL for both proving the verification conditions in the Isabelle proof assistant, as well as generating code that is operationally consistent with the verification semantics by constructing a proof that the generated code is correct with respect to the situations of the invariant diagram. We describe a mechanic translation of the transitions of an invariant diagram into a collection of mutually recursive functions and an associated correctness theorem stating that the value computed by the functions satisfies the final situation. We show that the proofs of the correctness theorem and the well-foundedness of the recursive functions can be built mechanically. The verification conditions are lemmas in the proofs. The collection of recursive functions is a refinement of the original invariant diagram, and is in a form that can be directly converted to executable code by Isabelle. This allows proof-producing compilation of invariant diagrams into any of the languages supported by Isabelle code generator. We illustrate our approach with a case study, and show that full proof automation can be achieved. This work is a step towards verified compilation of invariant diagrams.

Keywords: Program verification, Isabelle, Invariant-based programming

TUCS Laboratory
Software Construction Laboratory

1 Introduction

Invariant-based programming (IBP) is a programming methodology [3, 2, 1] in which a program is structured around its *situations* rather than the flow of control. A *situation* is an identified subset of the state space of the program—for instance, pre- and postconditions and loop invariants are situations. In IBP the situations are identified before the actual program code is written. In the first stage, the programmer establishes the overall situation structure of the program. Each situation is defined by a higher-order predicate on the state space, and the situations are structured according to their logical relationship. In the second stage, after all situations have been defined, the programmer goes on to add the *transitions*, the actual program code. Transitions are guarded program statements, connecting the situations. IBP is a *correct-by-construction* methodology. The programmer formally verifies each added transition correct with respect to the situations, before adding the next transition. The program and its proof of correctness are thus developed in lockstep.

The graph of situations and transitions is referred to as an *invariant diagram*. From an operational point of view, an invariant diagram can be seen as a transition system with unrestricted flow of control. Execution can start and terminate in any situation, rather than being restricted to single-entry and single-exit structures. From a mathematical point of view, an invariant diagram can be seen as a total correctness theorem, since the diagram is constructed together with the pre- and postconditions and invariants required to prove its correctness. The verification conditions (VCs) that must be proved are derived from the invariant diagram based on Hoare-like proof rules [4]. These proof rules have been shown to be sound and complete with respect to the operational semantics of invariant diagrams. The VCs can be proved by hand or in an automatic theorem prover. Socos is a tool for constructing invariant diagrams in the Eclipse IDE and automatically checking them in the PVS theorem prover and its associated SMT solver Yices [10].

In this paper, we study further the mechanization of IBP, this time in the Isabelle/HOL framework [19]. Isabelle is a generic meta-logical framework supporting multiple object logics; we assume the higher order logic HOL in this paper. We present a shallow embedding of invariant diagrams in Isabelle/HOL. This embedding has two objectives. Firstly, we are developing tool support for mechanical checking of the VCs derived from invariant diagrams based on Isabelle. Our goal is to build a VC generator using Isabelle as a back end, so that VCs could be proved either interactively in the proof assistant, or automatically using the state of the art decision procedures available in Isabelle. One such procedure, `sledgehammer`, which invokes multiple external state-of-the-art automatic provers simultaneously to discharge a proof goal, has proved highly useful in practical, day-to-day verification [8]. Our second objective is to compile invariant diagrams into executable code. Here we would like to take advantage of the new code generation framework for Isabelle [11], which can generate func-

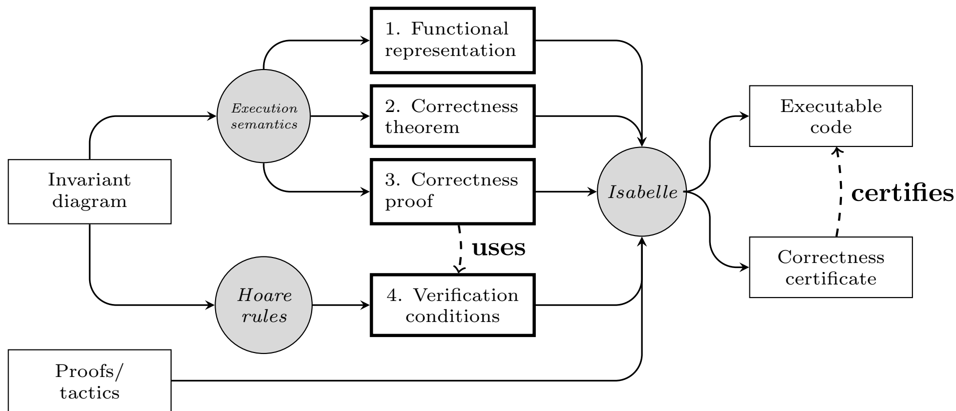


Figure 1: The VC generation and translation-validation compilation process

tional programs in multiple languages from recursive functions specified in the logic. The code generator is based on higher-order rewrite systems and supports translation into SML, OCaml, Haskell and Scala.

A fundamental problem of code generation in formal verification is that the compiler has to be trusted. Since the compilation of invariant diagrams is based on a different, lower-level, semantics than that of the VC generation, bugs in the compiler have the potential to nullify all the effort that went into verifying the VCs. One solution to this problem is to verify once and for all that the compiler is correct for all possible inputs. However, this is a substantial task. The technique outlined in this paper is based on *translation validation* [20]. Rather than verifying the compiler correct, we propose to extend the compiler to generate, for each compilation, not only the executable code but also a proof of its correctness. The proof will be checked in a separate validation phase during the code generation, and any compiler errors will be caught in this validation phase.

Figure 1 illustrates our verified code generation architecture for invariant diagrams. An invariant diagram is simultaneously translated into (1) an executable functional representation of the program; (2) a correctness theorem stating that the functional representation (1) implements the invariant diagram; (3) an Isabelle proof of theorem (2); and (4) the Hoare verification conditions associated with the program. The functional representation (1) is generated in a form that is readily executable. Theorem (2) states that (1) is a refinement of the original invariant diagram. The proof (3) certifies, based on the verification conditions (4) as lemmas, that the functional representation (1) satisfies the specification of the original program. The construction of this proof is mechanical. It is the responsibility of the programmer to discharge the verification conditions (4) in Isabelle (using automatic tactics as far as possible). Since the compilation is proof-producing, the trusted core consists only of the Isabelle/HOL proof system, and the Isabelle code generator. Hence, this allows verified compilation of invariant diagrams into any

of the languages supported by Isabelle without the need to verify the compiler.

Overview of paper. The sequel of the paper is structured as follows. We introduce some mathematical preliminaries in Section 2. We then introduce invariant diagrams, the underlying notation of IBP, in Section 3. Following this, we describe our embedding of an invariant diagram as an Isabelle theory in Section 4. Sections 5, 6 and 7 describe the correctness conditions—liveness, consistency and termination—given by the proof rules and how they are used to automatically discharge the conditions associated with the functional representation. In Section 8 we describe how to extract an executable program using the Isabelle code generator. Section 9 surveys related work in the field. We conclude the paper with a discussion of current and future work in Section 10.

2 Preliminaries

We use standard mathematical notations which maps directly to Isabelle notations. We use $A \to B$ for the type (or set) of functions from type A to type B, $f.a$ for function $f : A \to B$ applied to $a \in A$. Function type constructor associates to right, and the function application associates to left: $A \to B \to C$ is the same as $A \to (B \to C)$ and $f.a.b$ is the same as $(f.a).b$. We use $f \circ g$ for function sequential composition: $(f \circ g).x = f.(g.x)$. The identity function is denoted by $\mathsf{id} : A \to A$ and for all $x \in A$ we have $\mathsf{id}.x = x$. For two types A and B the *Cartesian product* of A and B is denoted by $A \times B$ and elements of $A \times B$ are called *pairs* and denoted by (a, b), where $a \in A$ and $b \in B$.

For separating the bounded variables from terms we use a small bullet ($\bullet$) and we use colon to separate variables or terms from their types: $(\forall\, x, y : \mathsf{Nat} \bullet x{+}y = y + x)$. We use lambda notation for constructing functions without naming them. $(\lambda\, x : \mathsf{Nat} \bullet x + 2)$ is the function which maps x to $x + 2$.

In general we use names starting with upper-case letters for denoting sets, and names starting with lower-case letters for constants and variables. We use sans-serif font for constants (true, false) and type constants (Bool, Nat) and we use normal mathematical italic font for variables (x, y) and type variables (A, $Index$).

Nat denotes the type of natural numbers and Bool denotes the type of the Booleans true and false. For Nat and Bool we assume the usual operations ($+, -, \ldots, \wedge, \neg, \ldots$). For $b \in \mathsf{Bool}$ and $x, y \in A$, the *if expression* ($\mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ y$) is equal to x when b is true and is equal to y otherwise.

Arrays with values from A are modeled as functions from Nat to A. When we are interested in finite arrays of size $n \in \mathsf{Nat}$, then we work only with the elements $a.0, \ldots, a.(n-1)$. For $a, b : \mathsf{Nat} \to A$ and $n \in \mathsf{Nat}$ the predicate $\mathsf{permutation}.n.a.b$ is true if the array b (of size n) is a permutation of array a (of size n), i.e. there exists a bijective function $f : \{0, ..., n-1\} \to \{0, ..., n-1\}$ such that $a \circ f = b$. For $a : \mathsf{Nat} \to A$ and $i, j \in \mathsf{Nat}$, $\mathsf{swap}.a.i.j$ denotes a new

array obtained from a by swapping the elements of indexes i and j.

$$\mathsf{swap}.a.i.j.k = \begin{cases} a.j & \text{if } k = i \\ a.i & \text{if } k = j \\ a.k & \text{otherwise} \end{cases}$$

A constant array is denoted by $[\![a_0, a_1, \ldots, a_{n-1}]\!]$, where

$$[\![a_0, a_1, \ldots, a_{n-1}]\!].k = \begin{cases} a_k & \text{if } k < n \\ \text{arbitrary fixed} & \text{otherwise} \end{cases}$$

Relations or *graphs* over a set A are sets of pairs with elements from A: $r \subseteq A \times A$. We denote the relations and the graphs of A by $\mathsf{Rel}.A$ and $\mathsf{Graph}.A$. We use the term relation when working with order like relations, and the term graph when we are interested in reachability properties. When we have a graph g over A we call the elements of A *vertexes*, and the pairs $(x, y) \in g$ *edges*.

A relation r on a set A is a *linear order* if it is a *total partial order* (*reflexive, antisymetric, transitive, and total*). In this paper we use the notation $LinOrd$ to denote a type variable with a linear order relation $\leq$ on it.

A relation r on A is *well founded* if it satisfies the *well founded induction property*:

$$\begin{array}{l} \mathsf{well_founded_induction}: \\ \quad (\forall P \bullet (\forall x \bullet (\forall y \bullet (y, x) \in r \Rightarrow P.y) \Rightarrow P.x) \Rightarrow (\forall x \bullet P.x)) \end{array}$$

We denote the set of well founded relations over A by $\mathsf{WF}.A$. A relation r is well founded, if and only if every *decreasing sequence* $a_1,\ a_2,\ \ldots\ ((a_{i+1},\ a_i) \in r)$ is finite.

If we have two relations r on A and r' on B then the *lexicographic composition* of r and r', denoted $r \times r'$, is a relation on $A \times B$ given by

$$((a,\ x),\ (b,\ y)) \in r \times r' \Leftrightarrow (a,\ b) \in r \vee (a = b \wedge (x,\ y) \in r').$$

If r and r' are well founded, then $r \times r'$ is also well founded.

For a relation r on A, the inverse of r is a relation on A, denoted r^-, where $(x,\ y) \in r^- \Leftrightarrow (y,\ x) \in r$, and the reflexive closure of r is also a relation on A, denoted $r^=$, where $(x,\ y) \in r^= \Leftrightarrow x = y \vee (x,\ y) \in r$.

For a type A, $\mathsf{List}.A$ denotes the type of *lists* with elements from A (finite sequences with elements from A). For $x \in A$ and $xs,\ ys \in \mathsf{List}.A$, $x\#xs \in \mathsf{List}.A$ is the list obtained from xs by adding the element x at the beginning, $xs@ys \in \mathsf{List}.A$ is the concatenation of the lists xs and ys, $\mathsf{hd}.xs \in A$ is the *head* of the list xs (the first element of xs), and $\mathsf{tl}.x \in \mathsf{List}.A$ is the *tail* of xs (the list obtained from xs by removing the first element). If xs is the empty list, then $\mathsf{hd}.x$ and $\mathsf{tl}.x$ are arbitrary but fixed elements of A and $\mathsf{List}.A$, respectively. We denote the empty

list by [] and a constant list with elements $a_1,\ a_2,\ \ldots,\ a_n$ by $[a_1,\ a_2,\ \ldots,\ a_n]$. On lists we can have an inductive definition of a function f by defining the function for the empty list and for the list $x\#xs$ assuming that f is already defined for xs.

For a list xs the set of elements of xs is given by $\mathsf{Set}.xs$:

$$\mathsf{Set}.[x_1,\ x_2,\ \ldots] = \{x_1,\ x_2,\ \ldots\}$$

For a function $f : A \to B$ and a list $xs : \mathsf{List}.A$ the function $\mathsf{map}.f.xs$ returns the list with elements given by function f applied to the elements of xs:

$$\mathsf{map}.f.[x_1,\ x_2,\ \ldots] = [f.x_1,\ f.x_2,\ \ldots].$$

If g is a graph over A, $s,\ s' \in A$, and $xs \in \mathsf{List}.A$ then the predicate $\mathsf{path}.g.s.s'.xs$ is true if xs is a *path* in g from s to s'. Formally path is defined by induction on xs:

$$\begin{array}{lcl} \mathsf{path}.g.s.s'.[] & = & \mathsf{false} \\ \mathsf{path}.g.s.s'.[x] & = & (x = s' \wedge s' = x) \\ \mathsf{path}.g.s.s'.(x\#y\#xs) & = & (x = s \wedge y = s' \wedge (x,y) \in g \wedge \mathsf{path}.g.y.s'.(y\#xs)) \end{array}$$

A path xs of a graph g from s to s' is nonempty if it contains at least one edge of g:

$$\mathsf{nonempty_path}.g.s.s'.xs = (\mathsf{path}.g.s.s'.xs \wedge (\exists\, u,\ v,\ ys \bullet xs = u\#v\#ys))$$

A path in a graph is *simple* if all its vertexes are distinct. A path in a graph is a *cycle* if the *initial* and *final* vertexes are the same ($s = s'$), and a cycle is *simple* if all its vertexes, except the first one, are distinct.

For a type $Index$ and a family of types $(A_i,\ i \in Index)$ the *disjoint union* of (A_i) is denoted by $(\bigoplus i : Index \bullet A_i)$. Formally the disjoint union of A_i is a type $A = (\bigoplus i : Index \bullet A_i)$ and a collection of injective functions $\mathsf{in}_i : A_i \to A$ which satisfy the property

$$(\forall\, a : A \bullet \exists!\, i : Index \bullet \exists\, b : A_i \bullet \mathsf{in}_i.b = a) \tag{1}$$

where $\exists!$ is the unique existential quantification. We should observe that in (1) b is also unique because in_i is injective. In other words for any element $a \in A$, there is a unique index i such that a is in the i component of the disjoint union A. For $a \in A$ we define the function $index.a \in Index$ which is the unique index i given by (1). And for $i \in Index$ and $a \in A$ we define the function $\mathsf{val}_i.a \in A_i$ to be the unique b from (1) if $i = index.a$ and it is arbitrary but fixed otherwise. In general we use the following naming rule. The name of the index set starts always with an upper-case letter, and the name the function computing the index of an element $a \in A$ is the same but with lower-case letters.

For A and $A_i \subseteq A$, we denote by $(\bigcup i \in Index \bullet A_i) \subseteq A$ the union of the sets A_i.

3 Invariant diagrams

We introduce invariant diagrams by a simple sorting program—shown in Figure 2—that will also serve as a running example throughout the paper. The invariant diagram shows an implementation of the *selection sort* algorithm for sorting an array of totally ordered elements in non-decreasing order. Selection sort extends in each iteration the already sorted region of the array by finding the minimum element in the remaining unsorted region.

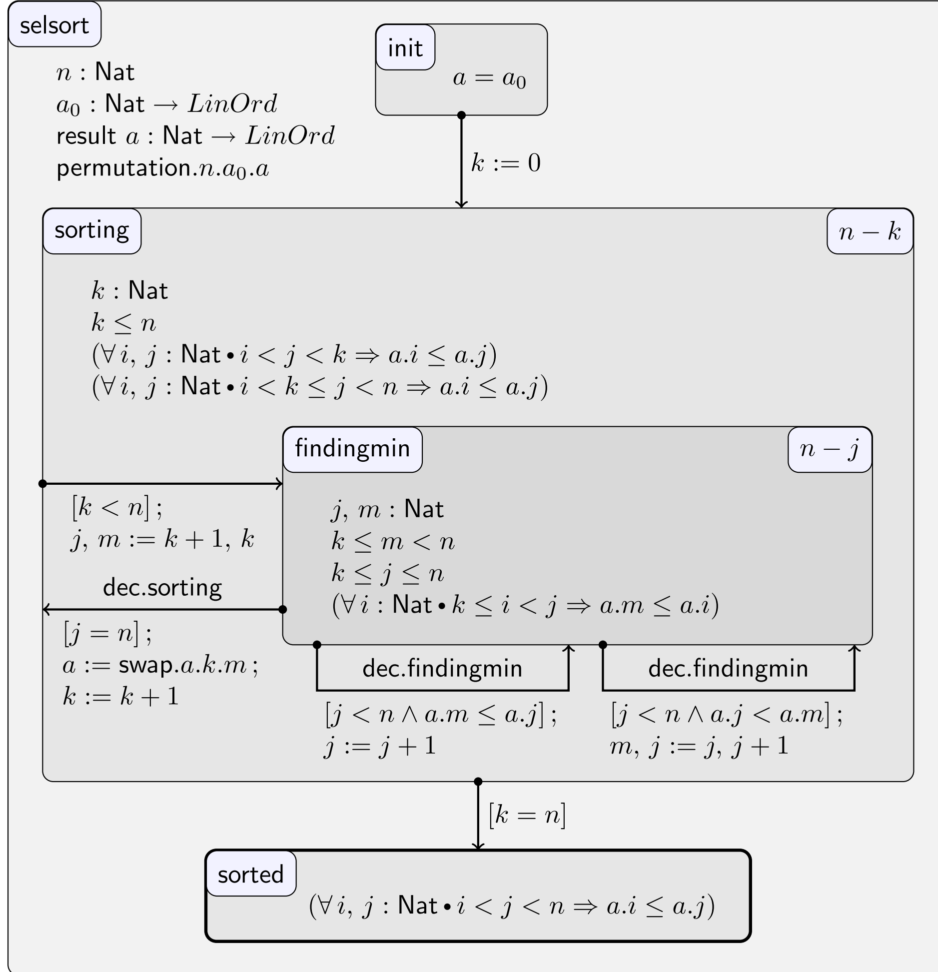


Figure 2: Invariant diagram representation of selection sort

The outermost situation, selsort, defines the global environment of selection sort. It introduces the variables n, a, a_0 representing respectively the length of the array, its elements, and its original value; and the invariant permutation.$n.a_0.a$. The *nested* situations of selection sort are init, sorting, findingmin, and sorted. A nested situation inherits the definitions and predicates of the outer situations. init

corresponds to the precondition of the program; in addition to the environment properties we know that a has the initial value a_0. The situation sorting corresponds to the loop invariant of the outer loop; in addition to the global environment, k is a new variable and all elements to the left of k are known to be sorted as well as smaller than or equal to the elements to the right of k. The predicate of findingmin is the conjunction of all predicates in the environment, sorting, and findingmin. It corresponds to the inner loop invariant, introducing the new variables j and m, and stating that $j \in \{k, \ldots, n\}$, m is the index of a minimum of a in the interval $\{k, \ldots, j-1\}$. sorted is the *final situation* of the program, stating that a is sorted. A diagram could have in general more than one final situation. In the diagram the final situations are marked with a thicker border. These invariants are sufficiently strong to prove selection sort correct.

For a diagram in general we denote by Sit the set of situations' labels, and we denote by SitSort the set of situations' labels of the sorting diagram, SitSort $=$ $\{$selsort, init, sorting, findingmin, sorted$\}$. The final situations of a diagram in general are denoted by Final, and the final situations of the sorting digram are denoted by FinalSort $= \{$sorted$\}$.

A transition from one situation to another or the same situation guarded by the predicate g and assigning the value(s) e to the program variable(s) x is written as $[g]\,;\,x := e$ on the transition arrow. Sequences of guards and assignments on the same transition are executed sequentially in the usual way. For simplicity we assume, without loss of generality, that all transitions have the form $[g]\,;\,x := e$ for some predicate g, some variable(s) x and some expression(s) e. The expressions e must correspond in types and number to variables x. By this assumption we do not loose the generality of transitions because every sequential composition of guards and assignments is equivalent to one of the form $[g]\,;\,x := e$. A transition $[g]\,;\,x := e$ is *enabled* in the current state if g is true. Unguarded transitions are always enabled. Execution of the diagram starts from any situation and follows the enabled transitions. If several transitions are enabled in the current situation, one is selected nondeterministically. If there are no enabled transitions execution terminates in the current situation. After each transition the predicate of the current situation must be satisfied by the current values of the program variables.

As we have seen already, a situation of an invariant diagrams inherits the variables from the other situations, and it may introduce new variables. For a situation $s \in$ Sit, we denote by x_s all these variables of s, and we denote by z_s only the new variables introduced by s and not those that are inherited. A transition $[g]\,;\,x := e$ from a situation s to a situation s' may assign values to all variables $x_{s'}$ and it may access in g and e all variables x_s of s. For example, in the sorting program, the transition from the situation sorting to situation findingmin assigns values to the new variables j and k defined in findingmin and it uses the values of k and n defined in situation sorting. As usually in Hoare logic, we need *specification variables* in invariant based programs. These are variables used for example to record at the end of a program the initial values of the variables modified by the

program. For the sorting program the variable a_0 is a specification variable which records the value of the array a at the beginning of sorting. We need this initial value of a in order to specify that after sorting a is a permutation of the original values. Formally for a invariant diagram the specification variables are denoted by SV and they are the variables that do not occur in the transitions. They are not assigned to, and they are not used in guards or expressions that are assigned to variables. The only specification variable of the sorting program is a_0. For a situation $s \in \mathsf{Sit}$ we denote by y_s all *execution variables* of s. These are all variables of s, which are not specification variables. Invariant diagrams are used to compute values of some certain variables. We can assume that the result of a diagram is given by the values of the variables defined by the final situations. However this approach is not satisfactory because in the final situation we may have also local variables or input variables that do not change during the execution. On a diagram we mark the result variables with the key word result. The only result variable of the sorting diagram is the array a. To avoid any ambiguities we require that all variables from a diagram have different names.

An invariant diagram is *correct* iff it is *consistent*, *live* and *terminating*. A diagram is consistent if each transition is *consistent*. A transition $[g]\,;\, x := e$ from a situation with predicate P to a situation with predicate Q is consistent iff

$$P \wedge g \implies Q[x \leftarrow e]$$

where $Q[x \leftarrow e]$ stands for the syntactic substitution of each free occurrence of the variable(s) x in Q by the expression(s) e.

Liveness means that the program terminates in one of the final situations. Liveness is established by checking for each non-final situation with predicate P that the guards $g_1, \ldots, g_n$ of the outgoing transitions satisfy:

$$P \implies g_1 \vee \cdots \vee g_n$$

Termination means that the program eventually reaches a situation with no enabled transitions, i.e., that the program cannot loop indefinitely. In contrast to consistency and liveness, verifying termination requires taking a global view of the program. The termination proof for selection sort is based on two separate *termination functions* (written in the upper right hand corner of the main recurring situation): $n-k$ for the outer loop ($Sorting \rightarrow FindingMin \rightarrow Sorting$) and $n-j$ for the inner loop ($FindingMin \rightarrow FindingMin$). Termination follows from the observation that both functions are bounded from below (by 0), that each iteration of the outer loop strictly decreases $n-k$, that each iteration of the inner loop strictly decreases $n-j$ without increasing $n-k$, and that there are no other loops in the diagram.

We refer to [3] for a general discussion of IBP and the correctness notions for invariant diagrams. For a formal treatment of the semantics of invariant diagrams, see [4].

4 Invariant diagrams in Isabelle/HOL

This section explains how an invariant diagram is embedded in an Isabelle theory. We describe first the encoding of situations, followed by the encoding of transitions.

4.1 Situation representation in Isabelle/HOL

We define the situations in Isabelle/HOL as *locales* [12]. A locale in Isabelle is a sub-theory with a local scope for constants (or parameters), assumptions, definitions, and theorems. A locale can *extend* other locales and we may have *locale interpretations*. A locale interpretation is an assignment of specific terms to the locale parameters. As a consequence of this assignment we should prove that the assumptions of the locales are true when the parameters are replaced by these terms. Figure 3 introduces the definition of two locales and an interpretation. We have a monoid locale that introduces two constants, a neutral element e and a multiplication operation $mult$, and assumes the monoid axioms. Next we have a group locale which extends monoid and introduces a new constant inv and assumes the group axioms. Within the monoid and group locales we can prove various theorems that follow from the axioms. Finally we have an interpretation $\texttt{Group}.0.+.-$ for the group locale where e, $mult$, and inv are replaced by the integer operations 0, $+$, and $-$. For this interpretation we have to prove that the concrete integer operations satisfy the group (and monoid) axioms. After this all theorems proved for the general group locale become available for the group of integers.

$$
\begin{array}{lll}
\textbf{locale}\ \texttt{Monoid} = & & \\
\quad \textbf{fixes} & e : A & \\
 & mult : A \to A \to A & \\
\quad \textbf{assumes} & (\forall\, x \bullet mult.e.x = mult.x.e = x) & \\
 & \ldots &
\end{array}
$$

$$
\begin{array}{lll}
\textbf{locale}\ \texttt{Group} = \texttt{Monoid}\, + & & \\
\quad \textbf{fixes} & inv : A \to A & \\
\quad \textbf{assumes} & (\forall\, x \bullet mult.x.(inv.x) = mult.(inv.x).x = e) & \\
\quad \textbf{theorem} & (\forall\, x,\, y \bullet inv.(mult.x.y) = mult.(inv.y).(inv.x)) & \\
 & proof\ \ldots &
\end{array}
$$

$$
\begin{array}{l}
\textbf{theorem}\ \texttt{Group}.0.+.- \\
\qquad proof\ \ldots
\end{array}
$$

Figure 3: Monoid and Group locales

In Isabelle the definition of a locale which extends previously defined locales with names $name_1 \ldots name_n$ is introduced with the key word **locale** followed by the name of the locale, the equality symbol, the names of the extended locales separated by the plus symbol ($name_1 + \ldots + name_n +$), constants (or parameters), assumptions, and theorems and definitions. The constants of the locales are introduced with the key word **fixes**. These constants are introduced together with their types, and the colon symbol is used to separate the name of a constant from its type. The assumptions are introduced with the keyword **assumes** and theorems are introduced with with the keyword **theorem**. Every theorem must have a formal proof written in Isabelle's proof language. Both assumptions and theorems can be named and their names can be used in proofs. Isabelle uses the keywords begin and end to mark the content of a locale. In this presentation we use indentation to serve the same purpose.

Locales are a good match for modeling situations. Each situation is modeled by a distinct locale. The locale's parameters or constants are used to model the situation variables. The locale's assumptions are used to model the predicates of the situation. Locales' extensions are used for modeling nesting of situations. Locale's interpretations are used to model the consistency proof obligations. The definitions and theorems of the locales may be used to structure the proofs of some of the more difficult proof obligations.

Each situation s, nested within situation s_0, and introducing new variables z_s and predicate P_s, introduces the locale L_s as follows:

$$\mathbf{locale}\ L_s = L_{s_0} +\ \mathbf{fixes}\ z_s\ \mathbf{assumes}\ P_s$$

In this context we assume that z_s is the list of new situation variables together with their types.

Selection sort situation locales are shown in Figure 4. Note that the situation locales are referenced only in the VCs and in the correctness proof produced by the compiler. They are not required for executing the transitions of the invariant diagram.

4.2 Functional representation in Isabelle/HOL

The executable part of an invariant diagram is represented in Isabelle as a collection of mutually recursive functions [13]. We introduce a function f_s for every situation $s \in \mathsf{Sit}$. The parameters of the function f_s are the execution variables y_s of s (all variables of s except the specification variables). The function f_s applied to some values v for the parameters y_s corresponds to the execution of the diagram, starting in s, with values v for the variables y_s. The result of f_s is a tuple with one component for every result variable of the diagram and an additional component that records the situation in which the termination has occurred. If $[g_1]\,;\, x_{s_1} := e_1$, $[g_2]\,;\, x_{s_2} := e_2$, $\ldots$ are all transitions from situation s, and if

locale `SelectionSort` =
  fixes n : Nat
    $a,\ a_0$: Nat $\rightarrow LinOrd$
  assumes permutation$.n.a.a_0$

locale `Init` = `SelectionSort` +
  assumes $a = a_0$

locale `Sorting` = `SelectionSort` +
  fixes k : Nat
  assumes $k \leq n$
    $(\forall\, i,\ j : \mathsf{Nat} \bullet i < j < k \Rightarrow a.i \leq a.j)$
    $(\forall\, i,\ j : \mathsf{Nat} \bullet i < k \leq j < n \Rightarrow a.i \leq a.j)$

locale `FindingMin` = `Sorting` +
  fixes j, m : Nat
  assumes $k \leq m < n$
    $k \leq j \leq n$
    $(\forall\, i : \mathsf{Nat} \bullet k \leq i < j \Rightarrow a.m \leq a.i)$

locale `Sorted` = `SelectionSort` +
  assumes $(\forall\, i,\ j : \mathsf{Nat} \bullet i < j < n \Rightarrow a.i \leq a.j)$

Figure 4: Locale representation of selection sort situations

for all i, $[g_i]\,;\, x_{s_i} := e_i$ is a transition between s and s_i, then the definition of the function f_s is

$$
\begin{aligned}
f_s.y_s \quad = \quad & \text{if } g_1 \text{ then } f_{s_1}.e_1 \\
& \quad \text{else if } g_2 \text{ then } f_{s_2}.e_2 \\
& \qquad \ldots \\
& \qquad \text{else } (res,\ s)
\end{aligned}
$$

Here we assume that the transitions assign values to all execution variables of the target situations, and we remind the reader that the expressions $g_1,\ g_2, \ldots$ and $e_1,\ e_2, \ldots$ may contain free only variables from y_s. When all guards are false this function returns the tuple res of result values and the current situation s. As we pointed out already $res = (res_1,\ res_2, \ldots)$ has a component for every result variable of the diagram. If the variable res_i is among the execution variables of s, then the value of res_i is the corresponding component of y_s, otherwise the value of res_i is arbitrary. We should also note here that the functional representation is a refinement of the execution mechanism of a diagram described earlier. If the execution of the diagram is in s and both guards g_1 and g_2 are true, then we could choose nondeterministically between the two transitions. However the functional representation always executes the first transition in this case.

Mutually recursive functions are introduced in Isabelle/HOL with the keyword **function**. We should introduce first the names and the types of the functions

function

$\mathsf{init_fun} : \mathsf{Nat} \to (\mathsf{Nat} \to LinOrd) \to (\mathsf{Nat} \to LinOrd) \times \mathsf{SitSort}$

$\mathsf{sorting_fun} : \mathsf{Nat} \to (\mathsf{Nat} \to LinOrd) \to \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat}) \times \mathsf{SitSort}$

$\mathsf{findingmin_fun} :$
$\quad \mathsf{Nat} \to (\mathsf{Nat} \to LinOrd) \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \to (\mathsf{Nat} \to LinOrd) \times \mathsf{SitSort}$

$\mathsf{sorted_fun} : \mathsf{Nat} \to (\mathsf{Nat} \to LinOrd) \to (\mathsf{Nat} \to LinOrd) \times \mathsf{SitSort}$

where

$\mathsf{init_fun}.n.a = \mathsf{sorting_fun}.n.a.0$

$\mathsf{sorting_fun}.n.a.k =$
$\quad$ if $k < n$ then $\mathsf{findingmin_fun}.n.a.k.(k+1)$
$\qquad$ else if $k = n$ then $\mathsf{sorted_fun}.n.a$
$\qquad$ else $(a, \mathsf{sorting})$

$\mathsf{findingmin_fun}.n.a.k.j.m =$
$\quad$ if $k < n \wedge j = n$ then $\mathsf{sorting_fun}.n.(\mathsf{swap}.a.n.k).(k+1)$
$\qquad$ else if $k < n \wedge j < n \wedge a.m \leq a.j$ then $\mathsf{findingmin_fun}.n.a.k.(j+1).m$
$\qquad$ else if $k < n \wedge j < n \wedge a.j < a.m$ then $\mathsf{findingmin_fun}.n.a.k.(j+1).j$
$\qquad$ else $(a, \mathsf{findingmin})$

$\mathsf{sorted_fun}.n.a = (a, \mathsf{sorted})$

Figure 5: Functional representation of selection sort

followed by the actual definitions. The functional representation of selection sort is shown in Figure 5.

The result of all functions of the sorting program is a pair of an array $a : \mathsf{Nat} \to LinOrd$ and a situation $s \in \mathsf{SitSort}$ because we have only a as a result variable. We skipped the introduction of a function for the situation selsort because the purpose of this situation is to provide the common variables and assumptions for all the other situations. The execution is not supposed to start in this situation, and it can never end here. In general, we can always skip the introduction of functions for situations that do not have incoming or outgoing transitions. We should also note that the functions are curried. To show that these functions are well defined, we should prove that the recursion always terminates. We postpone discussion of termination of the mutual recursion until Section 7.

The definition of the function findingmin_fun contains an additional guard $k < n$ which is always true when this function is called from sorting_fun. We need this additional guard when proving the termination of these functions.

The execution of the selection sort diagram starting from the init situation is equivalent to computing init_fun.$n.a$.

5 Consistency and liveness verification conditions

This section deals with the Isabelle representation of VCs for consistency and liveness. The consistency VCs and the liveness VC for all transitions with a common source situation are generated under the *context* of the source situation's locale. Isabelle's **context** blocks allow the user to add new theorems and definitions to locales defined earlier. For example if the group locale presented earlier together with the integer interpretation is in a library of facts, then we can extend the results of the group locale using the context block.

context `Group`
  theorem $inv_inv : (\forall x \bullet inv.(inv).x = x)$
    proof ...

After this extension the theorem inv_inv becomes available for groups as well as for the group of integers. The result of this context is the same as if we would introduce the theorem inv_inv in the locale group.

For each situation s with outgoing transitions to situations $s_1,\ s_2, \ldots$ having guards $g_1,\ g_2, \ldots$ and assignments $x_{s_1} := e_1,\ x_{s_2} := e_2, \ldots$, the consistency and liveness VCs are encoded as shown in the left hand column below:

context L_s
  theorem $\mathtt{to}_{s_1}$:
    assumes g_1
    shows $L_{s_1}.e_1$

  theorem `live`:
    shows $g_1 \vee \cdots \vee g_n$

context `Sorting`
  theorem `to_FindingMin`:
    assumes $k < n$
    shows
`FindingMin`.$n.a.a_0.k.(k+1).k$

  theorem `to_Sorted`:
    assumes $k = n$
    shows `Sorted`.$n.a.a_0$

  theorem `live`:
    shows $k < n \vee k = n$

Figure 6: Verification conditions

In the context of L_s if the guard g_1 is true, then the locale interpretation $L_{s_1}.e_1$ must be true. The theorem $\mathtt{to}_{s_1}$ can also be stated $g_1 \Rightarrow L_{s_1}.e_1$, but the advantage of using assumes and shows is that we can name the assumptions, and we can later

refer to them in proofs by their names. The liveness theorem states that the disjunction of the guards of all transitions from s must be true under the assumptions of L_s. The liveness VC must be added only to locales of non-final situations.

The right hand column shows the consistency and liveness VCs generated from the diagram for situation sorting. Actually proving that these VCs are all true is part of the verification effort. In practice, there are many theorems that need to be proved, so the task can be quite time-consuming. However, most of the theorems are quite trivial and are best tackled with automatic tactics first, with human attention needed only when this fails.

6 Consistency and liveness of the functional program

Next, we define consistency and liveness in terms of the functional representation of the program. Liveness and consistency properties are encoded in a locale and a theorem interpreting the locale. We first describe how they are derived. Subsequently, we describe how an automatic proof of this theorem is constructed. This proof certifies that the functional representation of the program is indeed consistent and live if the consistency and liveness VCs are discharged.

The program is consistent if, for all situations s and all initial values x_s of the variables of s satisfying $L_s.x_s$, the result (res', s') of the function f_s associated to s must be consistent, i.e., res' should satisfy the locale $L_{s'}$ associated to s'. Additionally, the program is live if s' is a final situation. Let $\mathsf{Final} = \{s_1, s_2 \ldots\} \subseteq \mathsf{Sit}$ be the set of final situations. We encode the consistency and liveness properties in the locale shown in the left hand column below:

locale `ConsLive` $=$
- **fixes** x_{Final}
- $s : \mathsf{Sit}$
- **assumes** $s \in \mathsf{Final}$
- $s = s_1 \Rightarrow L_{s_1}.x_{s_1}$
- $s = s_2 \Rightarrow L_{s_2}.x_{s_2}$
- $\ldots$

locale `ConsLiveSort` $=$
- **fixes** $n : \mathsf{Nat}$
- $a, a_0 : \mathsf{Nat} \rightarrow LinOrd$
- $s : \mathsf{SitSort}$
- **assumes** $s = \mathsf{sorted}$
- `Sorted`$.n.a.a_0$

The variables x_{Final} are all variables of all final situations. The constant s represents in this locale the situation in which the execution of the diagram terminates. The locale states that s must be a final situation and the final values must satisfy the locale corresponding to s. This means that the program is live, it would never terminate in a non-final situation. The right-hand column shows the `ConsLiveSort` locale for selection sort. In this case $s = \mathsf{sorted}$ and `Sorted`$.n.a.a_0$ must both hold.

Next we state the consistency and liveness of the functional representation of the program by interpretations of the `ConsLive` locale. Since execution of invari-

ant diagrams can start from any situation s, the locale must be satisfied by all states reached from each function f_s when called from a state satisfying L_s. We should prove for each s and x_s

$$L_s.x_s \wedge (res',\, s') = f_s.y_s \Rightarrow \texttt{ConsLive}.x_{\mathsf{Final}}[res \leftarrow res'].s' \tag{2}$$

where y_s are the execution variables of s, the variables res' are fresh, and $x_{\mathsf{Final}}[res \leftarrow res']$ stands for the syntactic replacement of the result variables res in x_{Final} with res'. I.e., if x_s satisfies the locale L_s, then the updated values $x_{\mathsf{Final}}[res \leftarrow res']$ computed by the function f_s must satisfy the locale ConsLive.

Next theorem is the correctness and liveness theorem for the selection sort.

Theorem 1.

$$\begin{array}{l}
\quad (\texttt{Init}.n.a.a_0 \wedge\ (a',\, s') = \mathsf{init_fun}.n.a \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\wedge\ (\texttt{Sorting}.n.a.a_0.k \wedge\ (a',\, s') = \mathsf{sorting_fun}.n.a.k \\
\qquad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\wedge\ (\texttt{FindingMin}.n.a.a_0.k.j.m \wedge\ (a',\, s') = \mathsf{findingmin_fun}.n.a.k.j.m \\
\qquad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\wedge\ (\texttt{Sorted}.n.a.a_0 \wedge\ (a',\, s') = \mathsf{sorted_fun}.n.a \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s')
\end{array}$$

This theorem states that regardless of the situation on which the program starts, if the locale of the starting situation is true and if the program terminates, then it terminates in the situation sorted, the computed array a' is sorted, and a' is a permutation of a_0. Although we may be interested only in starting the program in the initial situation, we need to state this theorem in this more general form to be able to prove it.

The Isabelle proof of Theorem 1 can be mechanically constructed based on the consistency and liveness verification conditions. The function definition 5 introduces an induction theorem that can be used to prove properties like Theorem 1. If we apply the induction theorem, then proving Theorem 1 is reduced to proving four properties, one for each function. The second property, associated to the function sorting_fun is

$$\begin{array}{l}
\qquad (k < n \wedge \texttt{FindingMin}.n.a.a_0.k.(k+1).k \\
\qquad\qquad \wedge\ (a', s') = \mathsf{findingmin_fun}.n.a.k.(k+1).k \\
\qquad\qquad\qquad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\wedge \\
\qquad (\neg k < n \wedge k = n \wedge \texttt{Sorted}.n.a.a_0 \wedge (a',\, s') = \mathsf{sorted_fun}.n.a \\
\qquad\qquad\qquad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\Rightarrow \\
\qquad (\texttt{Sorting}.n.a.a_0.k \wedge (a', s') = \mathsf{sorting_fun}.n.a.k \\
\qquad\qquad\qquad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s')
\end{array} \tag{3}$$

expanding the definition of sorting_fun we obtain three new properties corresponding to the three cases of sorting_fun:

$$\begin{array}{l}
\quad (k < n \wedge \texttt{FindingMin}.n.a.a_0.k.(k+1).k \\
\qquad \wedge\ (a',\ s') = \mathsf{findingmin_fun}.n.a.k.(k+1).k \\
\qquad\quad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\wedge \\
\quad (\neg k < n \wedge k = n \wedge \texttt{Sorted}.n.a.a_0 \wedge (a',\ s') = \mathsf{sorted_fun}.n.a \\
\qquad\quad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\Rightarrow \\
\quad (\texttt{Sorting}.n.a.a_0.k \wedge k < n \\
\qquad \wedge\ (a',\ s') = \mathsf{findingmin_fun}.n.a.k.(k+1).k \\
\qquad\quad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s')
\end{array} \tag{4}$$

and

$$\begin{array}{l}
\quad (k < n \wedge \texttt{FindingMin}.n.a.a_0.k.(k+1).k \\
\qquad \wedge\ (a',\ s') = \mathsf{findingmin_fun}.n.a.k.(k+1).k \\
\qquad\quad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\wedge \\
\quad (\neg k < n \wedge k = n \wedge \texttt{Sorted}.n.a.a_0 \wedge (a',\ s') = \mathsf{sorted_fun}.n.a \\
\qquad\quad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\Rightarrow \\
\quad (\texttt{Sorting}.n.a.a_0.k \wedge \neg k < n \wedge k = n \wedge (a',\ s') = \mathsf{sorted_fun}.n.a \\
\qquad\quad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s')
\end{array} \tag{5}$$

and

$$\begin{array}{l}
\quad (k < n \wedge \texttt{FindingMin}.n.a.a_0.k.(k+1).k \\
\qquad \wedge\ (a',\ s') = \mathsf{findingmin_fun}.n.a.k.(k+1).k \\
\qquad\quad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\wedge \\
\quad (\neg k < n \wedge k = n \wedge \texttt{Sorted}.n.a.a_0 \wedge (a',\ s') = \mathsf{sorted_fun}.n.a \\
\qquad\quad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s') \\
\Rightarrow \\
\quad (\texttt{Sorting}.n.a.a_0.k \wedge \neg k < n \wedge \neg k = n \wedge a' = a \wedge s' = \mathsf{sorting} \\
\qquad\quad \Rightarrow\ \texttt{ConsLiveSort}.n.a'.a_0.s')
\end{array} \tag{6}$$

The proof of these properties follow directly from the verification conditions presented in Figure 6 for the situation sorting. First two properties follow from the consistency VCs and the last one follows from the liveness VC.

7 Termination

An invariant diagram is terminating if no infinite computation can start from any situation. To establish the termination of the mutually recursive functions of a

diagram we should prove that a certain relation associated to the diagram is well founded. We call this relation the *termination relation* of the diagram and we will define it formally later. Proving that the termination relation is well founded would be very difficult because it involves arbitrary executions paths of the diagram. Instead, we handle termination analogously to consistency and liveness: by generating some VCs for transitions that are part of cycles in the diagram. Based on these VCs we automatically synthesize the proof that the termination relation is well-founded.

A situation s is a *cycle situation* if there is a path of transitions from s back to s. For each such situation we should provide in the diagram a *termination function* from the values of the variables to a set W with a well-founded relation $r \subseteq W \times W$. For selection sort, the cycle situations are sorting and findingmin and their termination functions are $n - k$: Nat and $n - j$: Nat, respectively. The well founded relation in this case is the normal order $<$ on Nat. For every cycle situation s, we should label some of the transitions as strictly decreasing the termination function of s, and we should prove as VCs that this is the case. This labeling should be such that all simple cycles from s to s contain at least one of these transitions. Moreover we should prove as VCs that all transitions that can occur on paths from s to s do not increase the termination function. The combined effect of these verification conditions is that we cannot have executions of the diagram which visits s infinitely often, and if we prove the same property for all cycle situations, then we cannot have infinite executions in general.

On the sorting diagram, the transition from findingmin to sorting is the only transition that should strictly decrease the termination function of sorting, and the two transitions from findingmin two findingmin are the transitions that should strictly decrease the termination function of findingmin. On the diagram these transitions are labeled by dec.sorting and dec.findingmin respectively. If we apply the technique described earlier for the situation findingmin, then we have to prove that at least one of the transition from sorting to findingmin and from findingmin to sorting strictly decreases the termination function of findingmin, because we have a simple cycle [findingmin, sorting, findingmin] and at least one of its transitions must strictly decrease the termination function of findingmin. However, this is not true. We could not solve this problem by taking the termination function of findingmin to be the pair $(n - k, n - j)$ with the lexicographic order, because this function is not defined for the transition from sorting to findingmin. We could replace the second component of this pair with the current situation to obtain a term which satisfies the desired conditions. However, here we consider a different solution. We order the cycle situations of the diagram s_1, $s_2, \ldots$ and we start with s_1. For s_1 we check that all simple cycles from s_1 to s_1 contain at least one strictly decreasing transition, and all other transitions of cycles from s_1 to s_1 are non increasing. After this we remove all incoming and outgoing transitions of s_1 and we repeat the process for $s_2, \ldots$ until we finish all cycle situations. Determining whether such an order exists, and if so constructing it, is accomplished by flow

graph analysis of the invariant diagram. For example, in selection sort the order [sorting, findingmin] satisfies this property. If an order cannot be found, no VCs will be emitted; in this case the programmer has to add decreasing transitions until an order is found. Applying this technique to the sorting diagram using the order [sorting, findingmin] results in the following verification conditions:

Situation sorting:

$$
\begin{array}{l}
\mathsf{sd_sorting_findingmin_sorting}: \\
\quad k < n \land j = n \Rightarrow (n - (k+1)) < (n - k) \\
\mathsf{ni_sorting_sorting_findingmin}: \\
\quad k < n \Rightarrow (n - k) \leq (n - k) \\
\mathsf{ni_sorting_findingmin_findingmin_a}: \\
\quad j < n \land a.m \leq a.j \Rightarrow (n - k) \leq (n - k) \\
\mathsf{ni_sorting_findingmin_findingmin_b}: \\
\quad j < n \land a.j < a.m \Rightarrow (n - k) \leq (n - k)
\end{array}
\tag{7}
$$

Situation findingmin:

$$
\begin{array}{l}
\mathsf{sd_findingmin_findingmin_findingmin_a}: \\
\quad j < n \land a.m \leq a.j \Rightarrow (n - (j+1)) < (n - j) \\
\mathsf{sd_findingmin_findingmin_findingmin_b}: \\
\quad j < n \land a.j < a.m \Rightarrow (n - (j+1)) < (n - j)
\end{array}
\tag{8}
$$

Naming of these proof obligations uses the following encoding: sd and ni stand for "strictly decreasing" and "non increasing", the first situation name is the situation giving the termination function, the second and the third situations are the starting and ending situation for the transition, and _a and _b are the indexes of the transitions when there are more than one transition between the same situations. The properties of strictly decrease or non increase should be proved only when the guards of the transitions are true. Ideally these properties should be proved assuming also the situations' predicates, but then they could not be used in the proof of the termination of the mutually recursive functions, unless we add these predicates as guards in the diagram. We have actually added the condition $k < n$ to the guards of the transitions starting from findingmin to be able to prove the termination.

In the remainder of this section we introduce more formally the concepts described so far and we show how these verification conditions are used mechanically to prove termination for the mutually recursive functions, i.e. the termination relation is well founded.

7.1 Well founded relation for function definition

For Sit a set of situations, the collection of mutually recursive functions $f_s,\ s \in \mathsf{Sit}$, where $f_s : A_{s,1} \to A_{s,2} \to \ldots \to B_s$, are defined by Isabelle internally using

one single recursive function

$$f : (\bigoplus s : \mathsf{Sit} \bullet A_s) \rightarrow (\bigoplus s : \mathsf{Sit} \bullet B_s)$$

where $A_s = A_{s,1} \times A_{s,2} \times \cdots$, which satisfies the conditions

$$\begin{array}{l} \quad (a \in A_s \Rightarrow f.(\mathsf{in}_s.a) \in \mathsf{in}_s.B_s) \\ \wedge \\ \quad f_s.a_1.a_2.\cdots = \mathsf{val}_s.(f.(\mathsf{in}_s.(a_1, a_2, \ldots))) \end{array}$$

The types $A_{s,1},\ A_{s,2}, \ldots$ are the types of the execution variables of situation s. The purpose of function f in this presentation is only to justify the introduction of the disjoint union $(\bigoplus s : \mathsf{Sit} \bullet A_s)$ of the types A_s, which is used for proving the termination of $(f_s,\ s \in \mathsf{Sit})$. An element a of $(\bigoplus s : \mathsf{Sit} \bullet A_s)$ represents the execution state of the diagram. The element a represents the situation $s = \mathsf{sit}.a$ as well as the values of the execution variables $\mathsf{val}_s.a \in A_s$. We remind the reader that the function $\mathsf{sit} : (\bigoplus s : \mathsf{Sit} \bullet A_s) \rightarrow \mathsf{Sit}$ returns for an element a the component s of the disjoint union to which a belongs.

For simplicity when $a = (a_1, a_2, \ldots) \in A_s$ we denote by $f_s.a$ the term $f_s.a_1.a_2.\cdots$. In our case the mutually recursive functions are tail recursive and the general form of the definition of f_s is

$$\begin{array}{rcl} f_s.y_s & = & \mathsf{if}\ g_1\ \mathsf{then}\ f_{s_1}.e_1 \\ & & \quad \mathsf{else\ if}\ g_2\ \mathsf{then}\ f_{s_2}.e_2 \\ & & \qquad \cdots \\ & & \qquad \mathsf{else}\ (res, s) \end{array}$$

In this definition y_s stands for the list of formal parameters of f_s, $g_1, g_2, \ldots$ are guards on the components of y_s, and $e_1 \in A_{s_1}, e_2 \in A_{s2}, \ldots$ are expressions on the components of y_s. In this definition we assume that $[g_1]\,;\, y_{s_1} := e_1$, $[g_2]\,;\, y_{s_2} := e_1$, $\ldots$ are all transitions from situation s. The *execution graph* of a diagram is a relation dgr on $(\bigoplus s : \mathsf{Sit} \bullet A_s)$ such that $(a,\ b) \in \mathsf{dgr}$ if and only if there is a transition from $s = \mathsf{sit}.a$ to $s' = \mathsf{sit}.b$ enabled for $\mathsf{val}_s.a$ and the result of the transition on $\mathsf{val}_s.a$ is $\mathsf{val}_{s'}.b$. More formally

$$\mathsf{dgr} = (\bigcup s : \mathsf{Sit} \bullet \mathsf{dgr}_s)$$

and

$$\mathsf{dgr}_s = \{(\mathsf{in}_s.y_s,\ \mathsf{in}_{s_1} e_1) \mid g_1\} \cup \{(\mathsf{in}_s.y_s,\ \mathsf{in}_{s_2}.e_2) \mid g_2\} \cup \ldots$$

For the sorting diagram the execution graph is given by

$$\begin{aligned}\mathsf{dgrsort} = \{&(\mathsf{in.sorting}.(n,a,k),\ \mathsf{in.findingmin}.(n,a,k,k,k+1) \mid k < n\} \\ \cup\ \{&(\mathsf{in.findingmin}.(n,a,k,j,m),\ \mathsf{in.sorting}.(n,(\mathsf{swap}.a.k.m),k+1)) \\ &\mid j = n\} \\ \cup\ \{&(\mathsf{in.findingmin}.(n,a,k,j,m),\ \mathsf{in.findingmin}.(n,a,k,j+1,m)) \\ &\mid j < n \wedge a.m \le a.j\} \\ \cup\ \{&(\mathsf{in.findingmin}.(n,a,k,j,m),\ \mathsf{in.findingmin}.(n,a,k,j+1,j)) \\ &\mid j < n \wedge a.j < a.m\} \\ \cup\ \{&(\mathsf{in.sorting}.(n,a,k),\ \mathsf{in.sorted}.(n,a,k)) \mid k = n\}\end{aligned}$$

In a diagram, an *execution path*, is a path in the graph dgr. In the sorting diagram, if we start the execution in situation sorting with $n = 4$, $a = [\![2,3,7,5]\!]$, and $k = 2$, then we may get the following execution path

$$\begin{aligned}[&\mathsf{in.sorting}.(4,a,2),\ \mathsf{in.findingmin}.(4,a,2,(j=3),(m=2)), \\ &\quad \mathsf{in.findingmin}.(4,a,2,4,3),\ \mathsf{in.sorting}.(4,[\![2,3,5,7]\!],3)]\end{aligned} \tag{9}$$

To prove in Isabelle that the mutual recursive functions f_s are terminating we should provide a well founded relation $<$ on $(\bigoplus s : \mathsf{Sit} \bullet A_s)$ and prove that

$$\begin{aligned}&(g_1 \Rightarrow \mathsf{in}_{s_1}.e_1 < \mathsf{in}_s.y_s) \wedge \\ &(\neg g_1 \wedge g_2 \Rightarrow \mathsf{in}_{s_2}.e_2 < \mathsf{in}_s.y_s) \wedge \\ &(\neg g_1 \wedge \neg g_2 \wedge g_3 \Rightarrow \mathsf{in}_{s_3}.e_3 < \mathsf{in}_s.y_s) \wedge \\ &\vdots\end{aligned} \tag{10}$$

If we take $<$ to be (dgr^{-1}) then it becomes trivial (mechanical) to prove the properties (10). We call dgr^{-1} the *termination relation* of the diagram. The challenge is to prove that this relation is well founded. In the next subsection we discuss the practical procedure for proving termination of invariant diagrams and we show how the resulting proof obligations can be used to mechanically prove that the termination relation is well founded.

7.2 Termination proof obligations

We first introduce the definitions of some graph concepts, and we introduce a theorem which reduces the well-foundness of the termination relation to some properties of execution paths of the diagram (paths of the graph dgr). Because the number of these paths is infinite, this reduction cannot be applied in practice, and we introduce next another theorem which reduces the well-foundness property to a finite number of VCs as discussed at the beginning of this section.

The predicate decrease : $\mathsf{Rel}.B \to (A \to B) \to \mathsf{List}.A \to \mathsf{Bool}$ is defined by

$$\begin{aligned}\mathsf{decrease}.r.t.[] &= \mathsf{true} \\ \mathsf{decrease}.r.t.[x] &= \mathsf{true} \\ \mathsf{decrease}.r.t.(x\#y\#xs) &= ((t.y, t.x) \in r^{=} \wedge \mathsf{decrease}.r.t.(y\#xs))\end{aligned}$$

where $\mathsf{decrease}.r.t.xs$ is true if for all consecutive elements x and y of xs, the function t applied to y is *smaller* or equal than t applied to x relatively to the *strict relation* r. The predicate $\mathsf{strictly_decrease} : \mathsf{Rel}.B \to (A \to B) \to \mathsf{List}.A \to \mathsf{Bool}$ is defined by

$$\begin{aligned}
&\mathsf{strictly_decrease}.r.t.[] &&= \mathsf{false}\\
&\mathsf{strictly_decrease}.r.t.[x] &&= \mathsf{false}\\
&\mathsf{strictly_decrease}.r.t.(x\#y\#xs) &&= (((t.y, t.x) \in r \wedge \mathsf{decrease}.r.t.(y\#xs))\\
& &&\quad \vee (t.y = t.x \wedge \mathsf{strictly_decrease}.r.t.(y\#xs)))
\end{aligned}$$

$\mathsf{strictly_decrease}.r.t.xs$ is true if the elements of xs are decreasing with respect to t and r and there are at least two consecutive elements x and y such that $t.y$ is strictly smaller than $t.x$ relatively to the strict relation r. For example the execution path (9) of the sorting diagram is strictly decreasing the term $t = n - k$ with respect to the natural order on Nat.

We what to ensure that all executions paths in the diagram are finite, and we ensure this property by requiring that all paths are strictly decreasing with respect to a well founded relation r and a termination term t. For $g \in \mathsf{Graph}.A$, $x, y \in A$, $r \in \mathsf{Rel}.W$, and $t : A \to W$ the predicate $\mathsf{strictly_decrease_all}.g.r.t.x.y$ is true if all nonempty paths from x to y in g are strictly decreasing with respect to r and t.

$$\begin{aligned}
&\mathsf{strictly_decrease_all}.g.r.t.x.y =\\
&\quad (\forall xs \bullet \mathsf{nonempty_path}.g.x.y.xs \Rightarrow \mathsf{strictly_decrease}.r.t.xs)
\end{aligned}$$

For $dgr \in \mathsf{Graph}.A$, $sit : A \to \mathsf{Sit}$, $ss \in \mathsf{List}.\mathsf{Sit}$,$r \in \mathsf{Rel}.W$, and $t : A \to W$, the predicate $\mathsf{strictly_decrease_loop}.ss.sit.dgr.r.t$ is true if all nonempty execution paths in the diagram dgr restricted to the situations in ss, from the situation $\mathsf{hd}.ss$ to $\mathsf{hd}.ss$ are strictly decreasing.

$$\begin{aligned}
&\mathsf{strictly_decrease_loop}.ss.sit.dgr.r.t =\\
&\quad (\forall x\, y \bullet sit.x = sit.y = \mathsf{hd}.ss\\
&\qquad \Rightarrow \mathsf{strictly_decrease_all}.(\mathsf{restrict}.ss.sit.dgr).r.t.x.y)
\end{aligned}$$

The function $sit : A \to \mathsf{Sit}$ returns the situation of a state $a \in A$ of the diagram. The list of situations ss plays the role of the situation ordering described at the beginning of this section, and $\mathsf{restrict}.ss.sit.dgr$ restricts the graph dgr to transitions from ss.

$$\mathsf{restrict}.ss.sit.dgr = \{(a,\, b) \mid (a,\, b) \in dgr \wedge sit.a \in \mathsf{Set}.ss \wedge sit.b \in \mathsf{Set}.ss\}$$

For $dgr \in \mathsf{Graph}.A$, $sit : A \to \mathsf{Sit}$, $ss \in \mathsf{List}.\mathsf{Sit}$, $r \in \mathsf{Sit} \to \mathsf{Rel}.W$, and $t : \mathsf{Sit} \to A \to W$ the predicate $\mathsf{strictly_decrease_dgr}.ss.sit.dgr.r.t$ is defined by induction on the list ss of situations. This predicate is true if for the situation $s = \mathsf{hd}.ss$ all nonempty execution paths starting and ending in s in the restricted

diagram are strictly decreasing; and the predicate is also true for the tail of ss.

$$\mathsf{strictly_decrease_dgr}.[].sit.dgr.r.t = \mathsf{true}$$

$$\begin{aligned}&\mathsf{strictly_decrease_dgr}.(s\#ss).sit.dgr.r.t = \\ &\quad \mathsf{strictly_decrease_loop}.(s\#ss).sit.dgr.(r.s).(t.s) \\ &\quad \wedge \mathsf{strictly_decrease_dgr}.ss.sit.dgr.r.t\end{aligned}$$

We should observe here that at every step in this recursive definition we remove from the diagram the transitions adjacent to the situation considered already.

Using this definition we are able to introduce a theorem that can be used to prove that the termination relation is well founded.

Theorem 2. *If* $dgr \in \mathsf{Graph}.A$, $sit : A \to \mathsf{Sit}$, $ss \in \mathsf{List.Sit}$, $r \in \mathsf{Sit} \to \mathsf{WF}.W$, *and* $t : \mathsf{Sit} \to A \to W$, *then*

$$\mathsf{strictly_decrease_dgr}.ss.sit.dgr.r.t \Rightarrow (\mathsf{restrict}.ss.sit.dgr)^{-1} \in \mathsf{WF}.A$$

The function t associates to every situation s a termination function $t.s$, a function from values of the execution variables to a set W with a well-founded relation $r.s$. In the beginning of this section we mentioned that we associate a termination function to every cycle situation. For non-cycle situations the termination function can be arbitrary because it will never be used. For the sorting diagram the well-founded relation is the natural order on Nat ($(x, y) \in \mathsf{less}.s \Leftrightarrow x < y$) and the termination function is given by

$$\begin{array}{lcl}\mathsf{term.sorting}.(\mathsf{in.sorting}.(n, a, k)) & = & n - k \\ \mathsf{term.sorting}.(\mathsf{in.findingmin}.(n, a, k, j, m)) & = & n - k \\ \mathsf{term.findingmin}.(\mathsf{in.findingmin}.(n, a, k, j, m)) & = & n - j \\ \mathsf{term}.s.x & = & 0 \text{ for all other } s \text{ and } x\end{array}$$

The definition of term.sorting.x has two main cases when x is a state from situation sorting and when x is a state from situation findingmin. In both situations the variables n and k are available and we require that the transition from findingmin to sorting strictly decreases term.sorting, which is true for this definition of term.sorting.

When Set.ss = Sit then restrict.ss.sit.dgr = dgr. In this case the Theorem 2 can be used to prove that the termination relation is well-founded. In the case of the sorting diagram we need to prove

$$\mathsf{strictly_decrease_dgr}.[\mathsf{init}, \mathsf{sorting}, \mathsf{findingmin}, \mathsf{sorted}].\mathsf{sit.dgrsort.less.term}$$

The order of the situations in the list [init, sorting, findingmin, sorted] is not arbitrary, but it is chosen such that the situation sorting comes before situation findingmin as we discussed at the beginning of this section. We first prove that all

execution paths from sorting to sorting are finite, and then we eliminate the transitions adjacent to sorting which ensures that all execution paths from findingmin to findingmin in the restricted diagram are also finite. The order in which init and sorted occur in this list is not relevant because there are no cycles containing these situations, and there are no verification conditions generated for them.

A problem with Theorem 2 is that strictly_decrease_dgr.$s.sit.dgr.r.t$ expands to an infinite number of proof obligations. This is due to the fact that there may be an infinite number of execution paths in the diagram dgr. We introduce later a new similar theorem which will allow proving that dgr^{-1} is well founded but using a finite number of proof obligations. The assumptions of this new theorem will be based on some functions which for diagrams with finite number of situations can be computed. The result of these computations will be the finite list of verification conditions presented at the beginning of this section. The main idea is to reduce reasoning about the set of execution paths to reasoning about the set of simple cycles in the *situation graph* associated to a diagram. The situation graph of a diagram is defined by

$$\mathsf{sdgr} = \{(s,\, s') \,|\, \text{there is a transition from } s \text{ to } s' \text{ in the diagram}\}$$

and for the sorting diagram we have

$$\begin{aligned}\mathsf{sdgrsort} = \{&(\mathsf{init},\, \mathsf{sorting}),\ (\mathsf{sorting},\, \mathsf{findingmin}),\ (\mathsf{findingmin},\, \mathsf{sorting}),\\ &(\mathsf{findingmin},\, \mathsf{findingmin}),\ (\mathsf{sorting},\, \mathsf{sorted})\}\end{aligned}$$

For a list xs and an element x, the function del.$x.xs$ deletes all occurrences of x from xs. For $g \in \mathsf{Graph}.A$, $xs \in \mathsf{List}.A$, and $x \in A$, the function next_vertex.$g.xs.x$ calculates the list of vertexes reachable from x in one step in the graph restrict.xs.id.g.

$$\begin{aligned}\mathsf{next_vertex}.g.[\,].x &= [\,]\\ \mathsf{next_vertex}.g.(y\#xs).x &= \mathsf{if}\ (x,y) \in g\ \mathsf{then}\ y\#(\mathsf{next_vertex}.g.xs.x)\\ &\ \ \mathsf{else}\ \mathsf{next_vertex}.g.xs.x\end{aligned}$$

For $g \in \mathsf{Graph}.A$, $zs, xs \in \mathsf{List}.A$, and $y \in A$, the function simple_path.$g.zs.xs.y$ calculates the list of all simple paths in the diagram restrict.zs.id.g from an element of xs to y.

$$\mathsf{simple_path}.g.zs.[\,].y = [\,]$$

$$\begin{aligned}&\mathsf{simple_path}.g.zs.(x\#xs).y =\\ &\quad(\mathsf{if}\ x \in \mathsf{Set}.zs\ \mathsf{then}\\ &\qquad(\mathsf{if}\ x = y\ \mathsf{then}\ [[x]]\ \mathsf{else}\ [\,])\\ &\qquad\quad @(\mathsf{map}.(\lambda u \bullet x\#u).\\ &\qquad\qquad(\mathsf{simple_path}.g.(\mathsf{del}.x.zs).(\mathsf{next_vertex}.g.(\mathsf{del}.x.zs).x).y))\\ &\quad\mathsf{else}\\ &\qquad[\,])\\ &\qquad\quad @(\mathsf{simple_path}.g.zs.xs.y)\end{aligned}$$

For $g \in \mathsf{Graph}.A$, $zs \in \mathsf{List}.A$, and $x \in A$, the function $\mathsf{simple_cycle}.g.zs.x$ calculates the list of all simple cycles in the diagram $\mathsf{restrict}.zs.id.g$ from vertex x to x.

$$\mathsf{simple_cycle}.g.zs.x = \mathsf{map}.(\lambda u \bullet x\#u).(\mathsf{simple_path}.g.zs.(\mathsf{next_vertex}.g.zs.x).x)$$

For $zs, xs \in \mathsf{List}.A$ the function $\mathsf{reachable}.g.zs.xs$ returns the list of reachable vertexes from the vertexes in xs in the graph $\mathsf{restrict}.zs.\mathsf{id}.g$.

$$\mathsf{reachable}.g.zs.[] = []$$

$$\begin{array}{l} \mathsf{reachable}.g.zs.(x\#xs) = \\ \quad \mathsf{if}\ x \in \mathsf{Set}.zs\ \mathsf{then} \\ \quad\quad x\#(\mathsf{reachable}.g.(\mathsf{del}.x.zs).((\mathsf{next_vertex}.g.(\mathsf{del}.x.zs).x)@xs)) \\ \quad \mathsf{else} \\ \quad\quad \mathsf{reachable}.g.zs.xs \end{array}$$

The set of reachable and co-reachable nodes from x in the graph $\mathsf{restrict}.zs.\mathsf{id}.g$ is defined by

$$\mathsf{core_re}.g.zs.x = \mathsf{Set}.(\mathsf{reachable}.g.zs.[x]) \cap \mathsf{Set}.(\mathsf{reachable}.(g^{-1}).zs.[x])$$

For $dgr \in \mathsf{Graph}.A$, $sit : A \to \mathsf{Sit}$, $s, s' \in \mathsf{Sit}$, $r \in \mathsf{Rel}.W$, and $t : A \to W$, the predicates
$\mathsf{decrease_edge}.sit.dgr.a.b.r.t$ and $\mathsf{strictly_decrease_edge}.sit.dgr.a.b.r.t$ calculate if all transitions from situation s to s' decrease and strictly decrease the function t with respect to r.

$$\begin{array}{l} \mathsf{decrease_edge}.sit.dgr.s.s'.r.t = \\ \quad (\forall x\, y \bullet (x, y) \in dgr \wedge sit.x = s \wedge sit.y = s' \Rightarrow (t.y, t.x) \in r^{=}) \end{array}$$

$$\begin{array}{l} \mathsf{strictly_decrease_edge}.sit.dgr.s.s'.r.t = \\ \quad (\forall x\, y \bullet (x, y) \in dgr \wedge sit.x = s \wedge sit.y = s' \Rightarrow (t.y, t.x) \in r) \end{array}$$

The predicates decrease_edge and strictly_decrease_edge are used to state the termination verification conditions in a compact manner. For example for the sorting diagram the predicate

$$\mathsf{strictly_decrease_edge.sit.dgrsort.findingmin.sorting.(less.sorting).(term.sorting)}$$

is equivalent to the verification condition sd_sorting_findingmin_sorting introduced earlier in this section.

The function $\mathsf{strictly_decrease_edge_path}.sit.dgr.ss.r.t$ calculates if there exists two consecutive situations s and s' in ss such that the condition

$\mathsf{strictly_decrease_edge}.sit.dgr.s.s'.r.t$ is true:

$$\mathsf{strictly_decrease_edge_path}.sit.dgr.[].r.t = \mathsf{false}$$

$$\mathsf{strictly_decrease_edge_path}.sit.dgr.[s].r.t = \mathsf{false}$$

$$\begin{array}{l}\mathsf{strictly_decrease_edge_path}.sit.dgr.(s\#s'\#ss).r.t = \\ \quad \mathsf{if\ strictly_decrease_edge}.sit.dgr.s.s'.r.t \mathsf{\ then\ true} \\ \quad \mathsf{else\ strictly_decrease_edge_path}.sit.dgr.(s'\#ss).r.t\end{array}$$

The predicate $\mathsf{strictly_decrease_loop_calc}.zs.sit.sdgr.dgr.r.t$ is true if all simple cycles from $\mathsf{hd}.zs$ to $\mathsf{hd}.zs$ in $\mathsf{restrict}.zs.\mathsf{id}.sdgr$ have a strictly decreasing edge, and all edges $(s, s') \in \mathsf{restrict}.zs.\mathsf{id}.sdgr$ where s and s' are reachable and correctable from $\mathsf{hd}.zs$, are non increasing. Formally we have:

$$\begin{array}{l}\mathsf{strictly_decrease_loop_calc}.zs.sit.sdgr.dgr.r.t = \\ \quad (\forall ss \bullet ss \in \mathsf{Set}.(\mathsf{simple_cycle}.sdgr.zs.(\mathsf{hd}.zs)) \\ \qquad \Rightarrow \mathsf{strictly_decrease_edge_path}.sit.dgr.ss.r.t) \\ \quad \wedge (\forall s\, s' \bullet s, s' \in \mathsf{core_re}.sdgr.zs.(\mathsf{hd}.zs) \wedge (s, s') \in sdgr \\ \qquad \Rightarrow \mathsf{decrease_edge}.sit.dgr.s.s'.r.t)\end{array}$$

All functions involved in the definition of strictly_decrease_loop_calc are defined in Isabelle, but they are also executable, i.e. the simplification mechanism of Isabelle is capable of calculating the result of strictly_decrease_loop_calc for concrete parameters.

For a function $sit : A \rightarrow \mathsf{Sit}$ and a graph $dgr \in \mathsf{Graph}.A$ we define the graph $sit.dgr \in \mathsf{Graph.Sit}$ by

$$sit.dgr = \{(s,\ s') \mid \exists a\ b \in A \bullet sit.a = s \wedge sit.b = s'\}$$

For a diagram the function sit maps the execution graph into a graph on situations and we have the property that sit.dgr $\subseteq$ sdgr. We have equality when every transitions of the diagram is enabled for some values of the variables.

Next theorem gives a method for calculating a finite set of proof obligations needed to show that all execution paths starting and ending in a situation $s = \mathsf{hd}.zs$ are strictly decreasing (finite).

Theorem 3. *If* $sit.dgr \subseteq sdgr$ *then*

$$\begin{array}{l}\mathsf{strictly_decrease_loop_calc}.zs.sit.sdgr.dgr.r.t \\ \quad \Rightarrow \mathsf{strictly_decrease_loop}.zs.sit.dgr.r.t\end{array}$$

Theorem 3 reduces proving that all execution cycles starting and ending in $\mathsf{hd}.zs$ are strictly decreasing if some finite verification conditions are true.

The final step in obtaining the verification conditions for the entire diagram is to inductively apply the predicate strictly_decrease_loop_calc.$zs.sit.sdgr.dgr.r.t$ for the tail of zs and so on.

$$\mathsf{strictly_decrease_dgr_calc}.[].sit.sdgr.dgr.r.t = \mathsf{true}$$

$$\begin{aligned}&\mathsf{strictly_decrease_dgr_calc}.(s\#zs).sit.sdgr.dgr.r.t = \\ &\qquad \mathsf{strictly_decrease_loop_calc}.(s\#zs).sit.sdgr.dgr.r.t \\ &\quad \wedge \mathsf{strictly_decrease_dgr_calc}.zs.sit.sdgr.dgr.r.t\end{aligned}$$

Theorem 4. *If* $dgr \in \mathsf{Graph}.A$, $sdgr \in \mathsf{Graph.Sit}$, $sit.dgr \subseteq sdgr$, $sit : A \to \mathsf{Sit}$, $ss \in \mathsf{List.Sit}$, $r \in \mathsf{Sit} \to \mathsf{WF}.W$, *and* $t : \mathsf{Sit} \to A \to W$, *then*

$$\mathsf{strictly_decrease_dgr_calc}.ss.sit.sdgr.dgr.r.t = (\mathsf{restrict}.ss.sit.dgr)^{-1} \in \mathsf{WF}.A$$

This final theorem can be applied now to a concrete diagram and it will produce a finite number of verification conditions. We have observed already that when $\mathsf{Set}.ss = \mathsf{Sit}$ then restrict.ss.sit.dgr $=$ dgr and if we apply this theorem for the sorting diagram we obtain that $\mathsf{dgrsort}^{-1}$ is well-founded if

$$\begin{aligned}&\mathsf{strictly_decrease_dgr_calc}.[\mathsf{init, sorting, findingmin, sorted}].\mathsf{sit} \\ &\quad .\mathsf{sdgrsort.dgrsort.less.term}\end{aligned} \tag{11}$$

is true, fact which can be simplified mechanically by Isabelle (by expanding the definitions) into

$$\mathsf{strictly_decrease_edge.sit.dgrsort.findingmin.sorting.(less.sorting).(term.sorting)}$$

$$\mathsf{decrease_edge.sit.dgrsort.sorting.findingmin.(less.sorting).(term.sorting)}$$

$$\mathsf{decrease_edge.sit.dgrsort.findingmin.findingmin.(less.sorting).(term.sorting)}$$

$$\begin{aligned}&\mathsf{strictly_decrease_edge.sit.dgrsort.findingmin.findingmin.(less.findingmin)} \\ &\quad \mathsf{.(term.findingmin)}\end{aligned}$$

These are exactly the termination verification conditions (7) and (8). We should note here that although we may have two or more transitions from a situation s to a situation s', if one of these transitions must strictly decrease a termination function t, then all transitions from s to s' must decrease t. In the sorting diagram this is the case for the two transitions from findingmin to findingmin.

8 Code generation

Isabelle/HOL comes with a built-in code generation framework based on term rewriting [11]. The framework translates recursive functions specified in HOL

into functional programs in an intermediate language. It currently supports Scala, SML, OCaml and Haskell target languages. The rewrites used during code generation can be extended with arbitrary equational theorems, as well as mappings from abstract datatypes into executable/more efficient datatypes (data refinement).

Generating code from the functional representation of an invariant diagram is straightforward. For instance, the Haskell rendition of selection sort is as follows:

```
data SitSort = Init | Sorting | Find_min | Sorted;

sorted_fun n a = (a, Sorted);

sorting_fun n a k =
  (if less_nat k n then find_min_fun n a k (plus_nat k one_nat) k
    else (if equal_nat k n then sorted_fun n a
           else (error "undefined", Sorting)));

find_min_fun n a k j m =
  (if less_nat k n && equal_nat j n
   then sorting_fun n (swap a k m) (plus_nat k one_nat)
   else (if less_nat k n && less_nat j n && less_eq (a m) (a j)
         then find_min_fun n a k (plus_nat j one_nat) m
         else (if less_nat k n && less_nat j n && less (a j) (a m)
               then find_min_fun n a k (plus_nat j one_nat) j
               else (error "undefined", Find_min))));

init_fun n a = sorting_fun n a Zero_nat;
```

Type declarations are omitted above for brevity. We note that as Haskell optimizes tail recursion, the program remains iterative throughout translation.

9 Related work

Translating the VCs of a program into the logic of a theorem prover, so called shallow embedding, is a well established technique in program verification. Some existing program verifiers employ Isabelle as a backend for VC generation. For instance, Jive is an interactive verifier for JML-annotated Java [9]. Boogie is an intermediate verification language and VC generator supporting multiple backends, including Isabelle/HOL [7]. These tools do not address verified compilation. A deep embedding (e.g., by representing programs as objects of an inductive datatype) allows reasoning about the embedded language itself in the theorem prover, but may require more effort to achieve high proof automation. For example, Simpl is an imperative language which is deeply embedded and fully formalized in Isabelle/HOL [22]. Deep embedding allows verifying the compiler in the theorem prover. Compilers from a subset of Java to bytecode [23], as well as from a subset of C to assembly [15], have been verified in Isabelle.

Various translation validation-based approaches have been proposed for proving safety properties and optimization correctness in low-level code. Necula and Lee introduced a compiler for the proof-carrying code framework [18], which produces certificates for type and memory safety. Blech and Poetzsch-Heffter have implemented a certifying compiler from a subset of C to MIPS that produces an Isabelle/HOL correctness proof [6]. The problem of translating function definitions in a theorem prover to executable code in a functional language prover has recently been addressed for HOL4 and Standard ML by Myreen and Owens [17].

Implementing flow charts as a collections of tail recursive functions is a well established technique; for instance, it was described already in 1962 by McCarthy [16].

Termination of transition systems is an active research topic. Size-change termination [14] is a general framework for proving termination by mapping infinite paths in a control flow graph to infinitely descending well-founded data values. Its main purpose is finding fully automatic termination proofs. Podelski and Rybalchenko [21] have shown that a program is terminating iff there exists a relation that contains the transitive closure of the transition relation as a subset (a *transition invariant*), and is a union of well-founded relations (a *disjunctively well-founded* transition invariant). Transition invariants allow more general termination arguments, but require reasoning about transitive transitions. An advantage of the method shown here is that a VC is emitted for each transition, allowing for an intuitive termination criterion (and, in our experience, it suffices for programs with well structured loops).

10 Conclusion and future work

We have in this paper described a translation validation approach to verified code generation for IBP, building on the fact that invariant diagrams have well-defined operational and verification semantics. We have shown how to mechanically translate an invariant diagram into an Isabelle/HOL theory consisting of the Hoare-like verification conditions, a set of mutually recursive functions implementing the transitions of the diagram, a consistency and liveness theorem, and a well-founded termination relation. The Hoare VCs that must be discharged by the programmer are *local* correctness conditions: they state that each transition establishes its target situation, that the disjunction of guards from each non-final situation is true, and that each loop decreases a termination function. The consistency and liveness theorem and the well-founded termination relation are *global* correctness conditions: they state that the functional representation of the whole program computes a value that satisfies the intended final situation. We have described how to mechanically construct the proof of the global correctness condition based on the local conditions as lemmas.

Our approach has the following advantages.

- Embedding invariant diagrams into Isabelle gives access to powerful tactics (such as `sledgehammer`) for discharging VCs. The functional representation can be directly translated into executable code by Isabelle's code generation framework.
- The trusted core of the system consists of only the theorem prover. Its code generator has to be trusted, but this gap is significantly smaller since the code generation is based on equational rewrite systems which are close to the proof theoretic framework of Isabelle (closing this gap, for HOL4, is addressed in [17]).
- No need to build a compiler for invariant diagrams. It is significantly easier to implement this method compared to constructing and formally verifying a compiler. The presence of a validating proof means that any errors in the compiler generating the functional representation of an invariant diagram will be detected.
- The embedding of invariant diagrams in HOL is shallow, allowing the program and its supporting theory to be developed together in a transparent fashion. This transparency is a practical advantage when constructing and verifying invariant-based programs. In our case studies, we have in fact used Isabelle theories as the main program source. However, specialized tool support could significantly simplify the program construction process.

We illustrated the approach with a case study. For the selection sort program we achieved almost full automation: the only VC requiring human interaction was that swapping two elements of an array maintains a permutation of the original array. Once this fact was proved and added to the Isabelle theory, bluntly applying `sledgehammer` worked well: all VCs were discharged by its default array of first order provers and SMT solvers.

There is room for much future work on this topic. We have not yet built the tool support for translating graphical invariant diagrams into Isabelle theories. We plan to automate this translation in our IBP environment Socos (see `www.imped.fi/socos`). Socos features a graphical diagram editor and exports VCs for the PVS theorem prover, but does not currently support compilation. Tool support for the approach presented here would allow assessment of its feasibility and scalability in practice. In particular, larger examples are required to identify potential bottlenecks in the proof construction and checking process. Also, we plan to support local definitions, another feature of Isabelle locales [5]. A situation could introduce local abbreviations and lemmas, which may be relevant for, e.g., discharging VCs related to that situation. Another research direction we plan to explore is more general termination proofs. A limitation in our current translation is that the termination proofs do not assume the loop invariant. Instead we added the assumptions necessary to prove termination to the guard manually. This step should be handled by the tool.

References

[1] Ralph-Johan Back. Program construction by situation analysis. Research Report 6, Computing Centre, University of Helsinki, Helsinki, Finland, 1978.

[2] Ralph-Johan Back. Invariant based programming revisited. Technical Report 661, Turku Centre for Computer Science, Turku, Finland, 2005.

[3] Ralph-Johan Back. Invariant based programming: Basic approach and teaching experiences. *Formal Aspects of Computing*, 21(3):227–244, 2009.

[4] Ralph-Johan Back and Viorel Preoteasa. Semantics and proof rules of invariant based programs. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 2011.

[5] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In *Types for proofs and programs (TYPES 2003)*, volume 3085 of *LNCS*, pages 34–50. Springer, 2004.

[6] Jan Olaf Blech and Arnd Poetzsch-Heffter. A certifying code generation phase. *Electron. Notes Theor. Comput. Sci.*, 190(4):65–82, November 2007.

[7] Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie: An interactive prover-backend for the Verifiying C Compiler. *Journal of Automated Reasoning*, 44(1–2):111–144, 2010.

[8] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.

[9] Á. Darvas and P. Müller. Formal encoding of JML Level 0 specifications in JIVE. Technical Report 559, ETH Zurich, 2007. Annual Report of the Chair of Software Engineering. 17 pages.

[10] Johannes Eriksson. *Tool-Supported Invariant-Based Programming*. Ph.d. thesis, Turku Centre for Computer Science, Finland, 2010.

[11] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*. Springer, 2010.

[12] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales - a sectioning concept for isabelle. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '99, pages 149–166, London, UK, UK, 1999. Springer-Verlag.

[13] Alexander Krauss. Defining Recursive Functions in Isabelle/HOL. Department of Informatics, Technische Universitt Mnchen, 2007.

[14] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, POPL '01, pages 81–92, New York, NY, USA, 2001. ACM.

[15] Dirk Leinenbach and Elena Petrova. Pervasive compiler verification – from verified programs to verified systems. *Electron. Notes Theor. Comput. Sci.*, 217:23–40, July 2008.

[16] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[17] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In *Proceedings of The 17th ACM SIGPLAN International Conference on Functional Programming*, 2012.

[18] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 33(5):333–344, May 1998.

[19] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[20] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, London, UK, UK, 1998. Springer-Verlag.

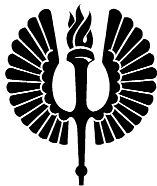
[21] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS '04: Proc. of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 32–41, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL.* PhD thesis, Technische Universität München, 2006.

[23] Martin Strecker. Formal verification of a java compiler in isabelle. In *Proceedings of the 18th International Conference on Automated Deduction*, CADE-18, pages 63–77, London, UK, UK, 2002. Springer-Verlag.

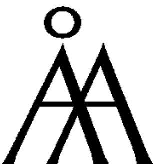
Turku Centre *for* Computer Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi



University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration

- Institute of Information Systems Sciences

ISBN 978-952-12-2791-2
ISSN 1239-1891