Under consideration for publication in Formal Aspects of Computing

Invariant Diagrams with Data Refinement

Viorel Preoteasa and Ralph-Johan Back

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland

Abstract. *Invariant based programming* is an approach where we start to construct a program by first identifying the basic situations (pre- and post-conditions as well as invariants) that could arise during the execution of the algorithm. These situations are identified before any code is written. After that, we identify the transitions between the situations, which will give us the flow of control in the program. Data refinement is a technique of building correct programs working on concrete data structures as refinements of more abstract programs working on abstract data types. We study in this paper data refinement for invariant based programs and we apply it to the construction of the classical Deutsch-Schorr-Waite graph marking algorithm. Our results are formalized and mechanically proved in the Isabelle/HOL theorem prover.

1. Introduction

Invariant based programming [Bac80b, Bac83, Bac08, BP11] is an approach to constructing correct programs where we start by identifying all basic situations (pre-conditions, post-conditions, and loop invariants) that could arise during the execution of the algorithm. These situations are determined and described before any code is written. After that, we identify the transitions between the situations, which together determine the flow of control in the program. The transitions are verified at the same time as they are constructed. The correctness of the program is thus established as part of the construction process.

We use a diagrammatic approach to describe invariant based programs, *(nested) invariant diagrams*, where situations are shown as (possibly nested) boxes and transitions are arrows between these boxes. We associate a collection of constraints with each situation box, and a sequence of simple statements with each transition arrow. Nesting expresses the information content of the situations: if situation B is nested within situation A, then B inherits the constraints of A. The control structure is secondary to the situation structure, and will usually not be well-structured in the classical sense, i.e., control is not necessarily expressible in terms of single- entry single-exit constructs. The invariant diagram shows explicitly all the information needed in order to verify that the program is correct: pre- and post-conditions, invariants, transitions, and termination functions.

We have been experimenting with teaching formal methods using invariant based programming for a number of years now, mainly in the form of introductory CS courses at university levels. The experiences have been good, the students learn quite easily how to construct programs that are correct by construction with this approach, and appreciate the added understanding that the approach brings to how a program works. The problems encountered have less to do with the invariants first approach than with the general problem of how to describe formally (in predicate calculus) situations that are intuitively well understood [Bac08].

The Figure 1 shows a simple example of an invariant based program, the factorial function, expressed as an invariant diagram.

There are three situations here, $init$, $loop$, and $final$. The situation $init$ declares the program variables n and x and restricts them to range over integers. In addition, we require that $n \geq 0$. The two other situations are nested inside $init$, which means that they inherit the program variables and constraints from the outer situation. Situation $final$ states that upon termination $x = n!$ must hold. The intermediate situation $loop$ declares an additional program variable i and restricts it to range over integers in the range 1 to $n + 1$. In addition, it requires that $x = (i - 1)!$

Correspondence and offprint requests to: Viorel Preoteasa, e-mail: viorel.preoteasa@abo.fi

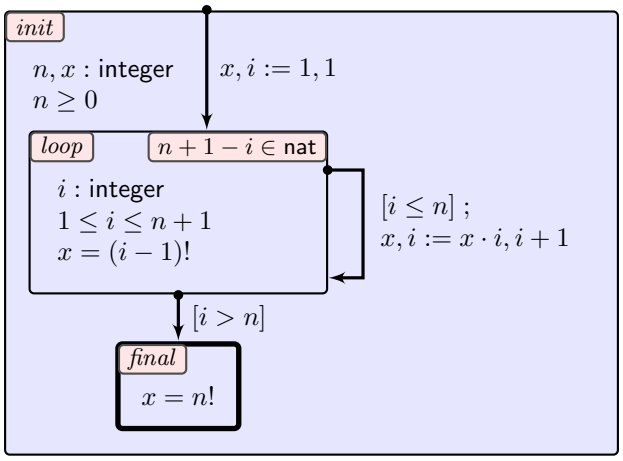

Figure 1. Factorial diagram (Factorial)

holds in this situation. There are three transitions in the diagram, one leading from *init* to *loop* providing initial values for x and i, one leading from *loop* back to itself that updates the variables x and i when $i \leq n$ holds, and one leading from *loop* to *final* that is taken when $i > n$ holds but which does not change any program variables. The transitions of the diagram can update the variables which are visible in the target situation. For example, in the factorial diagram, the transition from situation *init* to situation *loop* can update the variables x and i because x is defined in *init* and is also available in *loop*, and i is defined in *loop*.

The *loop* situation also gives a termination function, in the form *termination function* $\in W$, where W is some well founded set. In this case, the well founded set is the set of natural numbers nat and the termination function is $n + 1 - i$. The property $n + 1 - i \in$ nat must be provable from the constraints that hold in situation *loop* (in this case, this amounts to proving that $0 \leq n + 1 - i$ holds).

Execution of an invariant based program may start in any situation (not necessarily an initial situation), in a state that satisfies the constraints of the situation. One of the transitions that are enabled in this situation will be chosen. The transition is then executed, leading to new state in (possibly) another situation. There again one of the enabled transitions is chosen, and executed, and so on. In this way, execution proceeds from situation to situation. Execution terminates when a situation (not necessarily a final situation) is reached in a state for which there are no enabled transitions. Because the execution could start and terminate in any situation, invariant-based programs can be thought of as multiple entry, multiple exit programs. We may decide to identify some situations in an invariant based program as *initial situations* and some other situations as *final situation,* with the idea that execution should start in some initial situation and it should end in some final situation.

An invariant based program is *consistent*, if each transition *preserves* the situation constraints. This means that if we start execution in a situation A and in a state where the constraints of A are satisfied, and choose a transition that is enabled in A, then executing the transition will lead to some situation B (which could be A again) such that the resulting state satisfies the constraints associated with B. An invariant based program is *terminating*, if each execution of the program eventually terminates. For a given collection of final situations, an invariant based program is said to be *live* if termination only occurs in some final situation. The semantics and proof theory for invariant based programs are studied in detail in [BP11].

The idea of invariant based programming is not new, similar ideas were proposed in the 70's by John Reynolds [Rey78], Martin van Emden [VE79], and Ralph-Johan Back [Bac80b, Bac83], in different forms and variations. Dijkstra's later work on program construction also points in this direction [Dij76], where he emphasizes the formulation of a loop invariant as a central step in deriving the program code. However, Dijkstra insists on building the program in terms of well-structured (single-entry single-exit) control structures, whereas there are no restrictions on the control structure in invariant based programming. Central for these approaches is that the loop invariants are formulated before the program code is written. Eric Hehner [Heh79] was working along similar lines, but chose relations rather than predicates as the basic construct. Pnueli [Pnu05] used a graph structure with edges labeled by statements and nodes labeled by predicates to reason about correctness on imperative programs. His programs are also single entry, single exit and he does not use the nesting mechanism for defining the invariants.

The purpose of this paper is to study the use of *data refinement* when building invariant based programs. Data refinement [Hoa72, Bac80a, BvW00, DE99] is a technique of building correct programs working on concrete data structures as refinements of more abstract programs working on abstract data structures. The correctness of the final program follows from the correctness of the abstract program and from the correctness of the data refinement. The overall complexity of the correctness proof is usually lower when using data refinement than when the final program is developed directly on the concrete data structure. We will show how to adapt the basic idea of data refinement to the construction of invariant based programs. At the same time, we will extend the notion of nested invariant diagrams in a way that makes it easy to describe data refinement. On the theoretical side, this paper

extends the work described in [BP11] by including methods for carrying out data refinement. The data refinement theorems that we present here have all been proved mechanically in Isabelle/HOL [NPW02].

Data refinement is traditionally used as a technique for implementing data modules (or classes) with an abstract specification that describes the interface and the intended effect of the operations on the abstract data. The concrete implementation of the data module introduces a lot of detail that is not relevant for the way the data module is used. Data refinement shows that you can use the concrete implementation of the data module as if it was the abstraction itself, without having to change the interface or the way the data module is used in the rest of the program. Data refinement is thus a technique for enforcing information hiding in large, modularized programs. Our goals in this paper are, however, different. We are mainly interested in building complex algorithms, expressed in terms of concrete and often quite complex data structures. We develop the concrete algorithm through a sequence of refinement steps, starting from a rather abstract specification of the algorithm, which nails down some aspects that we want to achieve, but not all. Each refinement makes some aspect of the algorithm more concrete, until we have constructed an algorithm that satisfies the stated goals of efficiency and implementability. The difference to the information hiding view is that we are specifically interested in the final concrete algorithm and its concrete specification, expressed in terms of the concrete data structures manipulated by the algorithm.

We apply our technique to a larger case study, constructing the classical Deutsch-Schorr-Waite (DSW) [SW67, Knu97] marking algorithm for arbitrary graphs. The DSW algorithm marks all nodes in a graph that are reachable from a root node. The marking is achieved using only one extra bit of memory for every node. The graph is given by two pointer functions, left and right, which for any given node return its left and right successors, respectively. While marking, the left and right functions are altered to represent a stack that describes the path from the root to the current node in the graph. On completion the original graph structure is restored. We construct the DSW algorithm by a sequence of three successive data refinement steps. The entire development has been formalized in the Isabelle/HOL theorem prover. The main difference in our case study, as compared to the previous studies [MN05, Abr03], is that the whole refinement process is carried out using invariant diagrams. We also believe that the way we develop the algorithm by first proving a generalization of the algorithm significantly reduces the overall proof effort.

This paper is an extension of the paper [PB09]. We extend here the earlier version with a more general theory of data refinement for invariant based programs, and we instantiate this theory for the special diagrams used in [PB09]. We also introduce a new and more convenient way of handling program variables and program expressions in programming logic, by point-wise extended function application. We also significantly enhance the presentation of the Deutsch-Schorr-Waite case study, by adding more details about the refinement steps and by introducing a number of theorems about the correctness of the intermediate versions of the algorithm as well as about the refinement steps. A final theorem about the correctness of the DSW is also proved.

The paper is structured as follows. Section 2 presents the refinement calculus background. Section 3 introduces invariant diagrams and their proof theory. Section 4 introduces the data refinement of invariant diagrams. The DSW algorithm is constructed in Section 5. Section 6 presents some concluding remarks.

2. Refinement calculus background

We are going to formulate the proof theory of invariant diagrams in the refinement calculus. We therefore start with some essential notions that are going to be needed in the sequel.

We will here use period for function application, if $f : A \to B$ is a function from A to B and $x \in A$, then $f.x \in B$ is the application of f to x. The function application is assumed to be left associative. This means that we write $(f.e).\sigma$ as $f.e.\sigma$.

For two functions $f : X \to Y$ and $g : X \to Y \to Z$ we denote by $f[x \mapsto y] : X \to Y$ and $g[x, y \mapsto z] : X \to Y \to Z$, respectively, the functions given by

$$f[x \mapsto y].a = \begin{cases} y & \text{if } x = a \\ f.a & \text{otherwise} \end{cases}$$

$$g[x, y \mapsto z].a.b = \begin{cases} z & \text{if } x = a \wedge y = b \\ g.a.b & \text{otherwise} \end{cases}$$

We use also the notation $f[x_1 \mapsto y_1,\ \ldots,\ x_n \mapsto y_n]$ for $f[x_1 \mapsto y_1] \ldots [x_n \mapsto y_n]$.

2.1. Monotonic predicate transformers

We start by defining the notion of *monotonic predicate transformers,* on which the semantics of invariant based programs is based, and then continue with defining the different aspects of the correctness of program statements.

The type of Boolean values true and false is denoted by bool, and the boolean operations are $\wedge$, $\vee$, $\Leftarrow$, and $\neg$. We assume that we have a *state space* Σ. A state $\sigma \in \Sigma$ represents the values of the program variables. *Predicates*

(or *sets*) over Σ, denoted $\mathsf{pred}(\Sigma)$, are functions from $\Sigma \to \mathsf{bool}$. We denote by $\cup$, $\cap$, $\subseteq$, $\neg$ the *union, intersection*, *inclusion,* and *complement* of predicates respectively. When Σ is fixed we write only pred.

For a set of predicates $X \subseteq \mathsf{pred}$, $\bigvee X$ denotes the *disjunction* (*union*) of all predicates in X and $\bigwedge X$ denotes the *conjunction* (*intersection*) of all predicates in X. A predicate transformer S is *disjunctive* if for all $X \subseteq \mathsf{pred}$, $S.(\bigvee X) = \bigvee_{p \in X} S.p$. S is *conjunctive* if $S.(\bigwedge X) = \bigwedge_{p \in X} S.p$. S is *strict disjunctive* if $S.(\bigvee X) = \bigvee_{p \in X} S.p$ for all $X \neq \emptyset$, and S is *strict conjunctive* if $S.(\bigwedge X) = \bigwedge_{p \in X} S.p$ for all $X \neq \emptyset$.

In general we may have programs that start execution in a state Σ with some program variables and end in a state Σ' with other program variables. We model these *programs* as *monotonic predicate transformers,* denoted $\mathsf{mtran}(\Sigma, \Sigma')$, i.e., monotonic functions from $\mathsf{pred}(\Sigma') \to \mathsf{pred}(\Sigma)$. If $S \in \mathsf{mtran}$ and $q \in \mathsf{pred}(\Sigma')$, then $S.q \in \mathsf{pred}(\Sigma)$ are all states from which the execution of S terminates in a state satisfying the post-condition q. *Sequential composition* of programs, denoted $S\,;\,T$, is defined as the functional composition of monotonic predicate transformers, i.e. $(S\,;\,T).q = S.(T.q)$. Whenever we apply sequential composition or other operation to predicate transformers, we assume that they have the right types. In the sequential composition S is a program from Σ to Σ' and T from Σ' to Σ''. We denote by $\sqsubseteq$, $\sqcup$, and $\sqcap$ the point-wise extension of $\subseteq$, $\cup$, and $\cap$, respectively. The type mtran, together with the point-wise extension of the operations on predicates, forms a complete lattice. The partial order $\sqsubseteq$ on mtran is the *refinement relation* [BvW98, Bac80a]. The predicate transformer $S \sqcap T$ models *demonic choice* - the choice between executing S or T is arbitrary and cannot be influenced from outside.

Definition 1. If p and q are predicates and S is a program, then a *total correctness triple* $p \,\{\!|\, S \,|\!\}\, q$ is valid, denoted $\models p \,\{\!|\, S \,|\!\}\, q$, if and only if $p \subseteq S.q$.

We introduce a number of monotonic predicate transformers:

Assert	$\{\, p \,\}.q = p \cap q$
Assume	$[\, p \,].q = \neg p \cup q$
Demonic update	$[\, R \,].q.\sigma = (\forall \sigma' : R.\sigma.\sigma' \Rightarrow q.\sigma')$
Angelic update	$\{\, R \,\}.q.\sigma = (\exists \sigma' : R.\sigma.\sigma' \wedge q.\sigma')$
Magic	$\mathsf{magic}.q = \mathsf{true}$

The angelic update statement is disjunctive, the demonic update statement is conjunctive, the assert statement is disjunctive and strict conjunctive, and the assume statement is conjunctive and strict disjunctive.

2.2. Program expressions with point-wise function application

We assume that we have a number of program variables $x,\ y,\ z,\ \ldots$ of types $T_x,\ T_y,\ T_z,\ \ldots$. In this case the state space Σ is the Cartesian product of $T_x, T_y, T_z, \ldots$

$$\Sigma = T_x \times T_y \times T_z \times \ldots$$

and each state σ is some tuple $(a_x, a_y, a_z, \ldots,)$. Formally a program variable x is the corresponding projection function from Σ to T_x ($x = \mathsf{prj}_1$, $y = \mathsf{prj}_2$, $z = \mathsf{prj}_3$, $\ldots$). If $a \in T_x$ is a value then $\sigma[x := a] \in \Sigma$ is the state obtained from σ by changing the value of x to a. If $\sigma = (a_x,\ a_y,\ a_z, \ldots)$, then $\sigma[x := a] = (a,\ a_y,\ a_z, \ldots)$.

Lemma 2. If x and y are two distinct program variables, then the following properties hold:

$$x.(\sigma[x := a]) = a$$
$$y.(\sigma[x := a]) = y.\sigma$$

Program expressions of a type T are functions from $\Sigma \to T$. If $e : \Sigma \to T$ and $\sigma \in \Sigma$, then $e.\sigma$ is the value of expression e in state σ. If $e,\ f$ are program expressions and x is a program variable, then the *semantic substitution* of x with f in e, denote $e[x := f]$, is given by

$$e[x := f].\sigma = e.(\sigma[x := f.\sigma])$$

We point-wise extend all operators from values to program expressions. The point-wise extensions of plus and universal quantification are given by:

$$\begin{aligned} (e + f).\sigma &= e.\sigma + f.\sigma \\ (\forall x : b).\sigma &= (\forall a : (b[x := a].\sigma)) \end{aligned}$$

In the point-wise extension of the universal quantification a is a new variable which does not occur free in b. For example the value of expression $x + y$ in a state σ where $x.\sigma = 3$ and $y.\sigma = 4$ is 7.

Similarly we point-wise extend function application. For a program expression $e : \Sigma \to A$ and a function $f : A \to T$ we define the expression $f.e : \Sigma \to T$ by

$$(f.e).\sigma := f.(e.\sigma)$$

Here only the second argument of application, e, is applied to the state σ. For a program expression $e : \Sigma \to A$ and a function $g : \Sigma \to A \to T$ we define the expression $g.e : \Sigma \to T$ by

$$(g.e).\sigma := (g.\sigma).(e.\sigma)$$

Here both the first and the second argument of application are applied to the state σ. In the sequel we will assume that the extended application operation is left associative, like the standard application operation. This means that we write $(f.e).\sigma$ as $f.e.\sigma$ and $(g.e).\sigma$ as $g.e.\sigma$.

The big advantage of using point-wise extended application is that we can be explicit about which program variables our predicates depend upon. For example if $q : T_x \to T_y \to \mathsf{bool}$ is a function with two parameters, then with point-wise extended application, we have that

$$q.x.y = (\lambda\sigma : ((q.x).y).\sigma) = (\lambda\sigma : ((q.x).\sigma).(y.\sigma)) = (\lambda\sigma : q.(x.\sigma).(y.\sigma))$$

Here we use both forms for point-wise extended application in calculating the corresponding lambda function. The expression $q.x.y$ is a predicate on Σ. The expression shows explicitly that the predicate only depends on the program variables x and y. Moreover, if we substitute program variable x by expression e in $q.x.y$, we get $q.e.y$:

$$(q.x.y)[x := e] = q.e.y$$

The point-wise extension of the function application turns out to be very convenient when calculating with program expressions. For function application, we have that $(q.x)[x := e] = q.e$, i.e., the substitution of x by e in $q.x$ is $q.e$. This same property holds for point-wise extended function application, as shown above. We also get a generalization of the point-wise extension of all other operators. For example we can use the same construction as we used for q above to build $x + y : \Sigma \to \mathsf{integer}$ using $+ : \mathsf{integer} \to \mathsf{integer} \to \mathsf{integer}$;

$$+.x.y.\sigma = ((+.x).\sigma).(y.\sigma) = +.(x.\sigma).(y.\sigma) = x.\sigma + y.\sigma$$

For us, the most important fact is that we can express Hoare correctness triples in a manner which is easy to read and understand. For example if $p.a = (0 < a)$ and $q.a.b = (0 \le a < b)$, and if x, y are some integer program variables, then we can state the correctness of the program $x := y - 1$ as

$$\models p.y \,\{\![\, x := y - 1 \,]\!\}\, q.x.y \tag{1}$$

instead of having to write this as

$$\models (\lambda a, b : p.b) \,\{\![\, x := y - 1 \,]\!\}\, (\lambda a, b : q.a.b) \tag{2}$$

where we also need to remember that Σ has two components for x and y.

Applying the rule for the assignment statement in (1) results in the proof obligation

$$p.y \subseteq q.(y - 1).y$$

which is true if and only if

$$\begin{aligned}
&(\forall\sigma : p.y.\sigma \Rightarrow q.(y-1).y.\sigma) \\
&= (\forall\sigma : p.(y.\sigma) \Rightarrow q.(y.\sigma - 1).(y.\sigma)) \\
&= (\forall(a,b) : p.(y.(a,b)) \Rightarrow q.(y.(a,b) - 1).(y.(a,b))) \\
&= (\forall a, b : p.b \Rightarrow q.(b-1).b)
\end{aligned}$$

2.3. Assignment statements

We define some more specific program statements as special cases of the demonic and angelic update statements. These are the *demonic assignment statement* $[x := x' \mid b.x'.y.z]$, the *angelic assignment statement* $\{x := x' \mid b.x'.y.z\}$, and the *assignment statement* $x := e$. In the demonic and angelic assignment statements we assume that the starting state has only the variables $y,\ z$ and the final state has only the variables $x,\ z$. The variable x is calculated using the values of y and z according to the relation b, and z is left unchanged. We define these as follows:

Demonic assignement	$[\, x := x' \mid b.x'.y.z \,]$	$=$	$[\, \lambda\sigma, \sigma' : z.\sigma = z.\sigma' \wedge b.(x.\sigma').(y.\sigma).(z.\sigma) \,]$
Angelic assignement	$\{\, x := x' \mid b.y.z.x' \,\}$	$=$	$\{\, \lambda\sigma, \sigma' : z.\sigma = z.\sigma' \wedge b.(x.\sigma').(y.\sigma).(z.\sigma) \,\}$
Assignment	$(x := e)$	$=$	$[\, x := x' \mid x' = e \,]$

The contexts in which these statements are used define precisely the initial variables ($y,\ z$) and the final variables ($x,\ z$).

Lemma 3. If $b : T_x \to T_y \to T_z \to \mathsf{bool}$, $q : T_x \to T_z \to \mathsf{bool}$, and $p : T_x \to \mathsf{bool}$, then the following properties are true

$$\begin{array}{lcl} [\, x := x' \mid b.x'.y.z \,].(q.x.z) & = & (\forall x' : b.x'.y.z \Rightarrow q.x'.z) \\ \{\, x := x' \mid b.x'.y.z \,\}.(q.x.z) & = & (\exists x' : b.x'.y.z \wedge q.x'.z) \\ (x := e).(p.x) & = & p.e \end{array}$$

(Note that we use the point-wise extended quantifiers on the left).

Proof. We show the proof only for the first property. The others are similar.

$$\begin{array}{ll} \bullet & [\, x := x' \mid b.x'.y.z \,].(q.x.z).\sigma \\ = & \{\text{Definition}\} \\ & [\, \lambda\sigma, \sigma' : z.\sigma = z.\sigma' \wedge b.(x.\sigma').(y.\sigma).(z.\sigma) \,].(q.x.z).\sigma \\ = & \{\text{Definitions}\} \\ & (\forall \sigma' : z.\sigma = z.\sigma' \wedge b.(x.\sigma').(y.\sigma).(z.\sigma) \Rightarrow q.(x.\sigma').(z.\sigma')) \\ = & \{\text{Logic}\} \\ & (\forall \sigma' : (\exists a : x.\sigma' = a) \wedge z.\sigma = z.\sigma' \wedge b.(x.\sigma').(y.\sigma).(z.\sigma) \Rightarrow q.(x.\sigma').(z.\sigma)) \\ = & \{\text{Logic}\} \\ & (\forall a, \sigma' : x.\sigma' = a \wedge z.\sigma = z.\sigma' \wedge b.a.(y.\sigma).(z.\sigma) \Rightarrow q.a.(z.\sigma)) \\ = & \{\text{Logic}\} \\ & (\forall a : (\exists \sigma' : x.\sigma' = a \wedge z.\sigma = z.\sigma') \wedge b.a.(y.\sigma).(z.\sigma) \Rightarrow q.a.(z.\sigma)) \\ = & \{\text{Using the witness } (a,\ z.\sigma) \text{ for } \sigma' \text{ which has only two components, for } x \text{ and } z\} \\ & (\forall a : b.a.(y.\sigma).(z.\sigma) \Rightarrow q.a.(z.\sigma)) \\ = & \{\text{Definitions}\} \\ & (\forall x' : b.x'.y.z \Rightarrow q.x'.z).\sigma \end{array}$$

In the theory part we talk about program variables x and y and we assume that they are distinct. In practical examples we will use tuples of concrete variables instead of x and y. In this case the condition that they are distinct is replaced by the conditions that the tuples do not have common concrete variables.

2.4. Guards

Definition 4. For a monotonic predicate transformer $S \in \mathsf{mtran}$ we introduce the *guard* of S by $\mathsf{grd}.S = \neg(S.\mathsf{false})$.

Theorem 5. The following properties are true.

$\mathsf{grd}.(x := e) = \mathsf{true}$
$\mathsf{grd}.[\, p \,] = p$
$\mathsf{grd}.[\, x := x' \mid b \,] = (\exists x' : b)$
$\mathsf{grd}.(S_1 \sqcap S_2) = \mathsf{grd}.S_1 \cup \mathsf{grd}.S_2$

3. Proof theory for invariant diagrams

We now continue with developing the proof theory of invariant based programs. The presentation here is a summary of the work described in [BP11].

3.1. Invariant diagrams

An *invariant diagram* is a directed graph where nodes are labeled with *invariants* (predicates) and edges are labeled with *program statements* (monotonic predicate transformers). The nodes of the invariant diagram are called *situations* and the edges are called *transitions*.

Let I be a nonempty set of indexes. Formally, an invariant diagram is a tuple (P, T) where $P : I \to \mathsf{pred}$ are the *invariants* and $T : I \times I \to \mathsf{mtran}$ are the *transitions*. We refer to T as a *transition diagram.* The guard of a transition diagram in a situation $i \in I$, $\mathsf{grd}.T.i \in \mathsf{pred}$ is the disjunction of the guards of all transitions from i:

$$\mathsf{grd}.T.i = \bigvee_{j \in I} \mathsf{grd}.T.i.j$$

We denote by Magic the diagram in which all transitions are magic:

$\mathsf{Magic} = (\lambda i, j : \mathsf{magic})$.

Figure 1 is a representation for the transition diagram Factorial given by:

$$\begin{array}{lcl} \mathsf{Factorial} & = & \mathsf{Magic}[\\ & & \quad (\mathit{init}, \mathit{loop}) \mapsto (x, i := 1, 1) \\ & & \quad (\mathit{loop}, \mathit{loop}) \mapsto ([i \leq n]\,;\, x, i := x \cdot i, i + 1) \\ & & \quad (\mathit{loop}, \mathit{final}) \mapsto [i > n]] \end{array}$$

and for the invariants:

$$\begin{array}{ll} P.\mathit{init} & = (n \geq 0) \\ P.\mathit{loop} & = (n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1) \\ P.\mathit{final} & = (n \geq 0 \wedge x = n!) \end{array}$$

In this example the set I of situations is $\{\mathit{init},\ \mathit{loop},\ \mathit{final}\}$. A transition from situation i to situation j is assumed to be magic (the transition which is never enabled) if it is not drawn explicitly in the diagram.

The execution of an invariant diagram may start in any situation $i \in I$, and then non-deterministically choose some enabled transition from i leading to some new situation $j \in I$. The execution continues in this way as long as transitions are enabled. The execution terminates when a situation i is reached where no transitions are enabled. In [BP08, BP11], we have introduced operational semantics and predicate transformer semantics for invariant based programs and we proved their equivalence. We have also introduced consistent and complete proof rules for invariant diagrams. We recall these proof rules below.

An *executable invariant diagram* is one in which all statements in the diagram are equivalent to the form $[\,p\,]\,;\, x := e$, where e and p are ordinary program expressions. They should thus not contain quantifiers or specification functions which are not part of the target programming language.

Very often in examples it is convenient to draw more than one transition between two situations i and j. We interpret these as standing for a single transition that is the demonic choice of all transitions between i and j.

3.2. Correctness of invariant diagrams

Informally, a transition diagram $T : I \times I \to \mathsf{mtran}$ is *totally correct* with respect to the initial predicates $P : I \to \mathsf{pred}$ and final predicates $Q : I \to \mathsf{pred}$, denoted $P \,\{\!|\, T \,|\!\}\, Q$, if for all initial states σ and situations i for which $P.i.\sigma$ is true, the execution always terminates, and $Q.j.\sigma'$ is true for the termination state σ' and termination situation j. This notion is defined more precisely in [BP11]. For this paper, we do not need the exact semantic definition of total correctness of transition diagrams, the proof rule established in [BP11] will be sufficient.

Let us first give a proof rule for establishing a special case of total correctness, called *strict total correctness*. This is ordinary total correctness, but the pre- and post-conditions have to be of a specific form. The proof rule is the following:

$$\frac{\models X.w.i \,\{\!|\, T_{i,j} \,|\!\}\, X_{<(i,w)}.j, \quad \text{for all } i, j, w}{\vdash_s \bigvee X \,\{\!|\, T \,|\!\}\, \bigvee X \cap \neg \mathsf{grd}.T} \tag{3}$$

where $X : W \to I \to \mathsf{pred}$, W is a set, $<$ is a well ordered relation on $I \times W$, and

$$\left(X_{<(i,w)}\right).j \quad = \quad \bigvee_{(j,v)<(i,w)} X.v.j \quad \text{and} \quad \bigvee X \quad = \quad \bigvee_{w \in W} X.w$$

In the proof rule (3) we used for transitions valid Hoare triples ($\models p \,\{\!|\, S \,|\!\}\, q$) instead of provable Hoare triples ($\vdash p \,\{\!|\, S \,|\!\}\, q$). We used valid Hoare triples in order to avoid the introduction of proof rules for transitions, which are not relevant for this paper. The equivalence between valid Hoare triples and provable Hoare triples was studied in [BP11].

The proof rule (3) reduces the correctness of a transition diagram to total correctness assertions for all transitions, which can be proved in the ordinary refinement calculus framework. We write the index s at the proof symbol, to indicate that the total correctness was established with this specific rule. Termination of execution of the diagram is ensured here, because every transition must decrease a variant with respect to the well ordered relation $<$. Having a well ordered relation on both the set of situations and some other set W results in a rule which is easier to apply in practical examples, as we will show on the factorial example.

Total correctness of a diagram in general is proved by combining the traditional rule of consequence with the rule for proving strict total correctness:

$$\frac{P \subseteq P' \quad Q' \subseteq Q \quad \vdash_s P' \,\{\!|\, T \,|\!\}\, Q'}{\vdash P \,\{\!|\, T \,|\!\}\, Q} \tag{4}$$

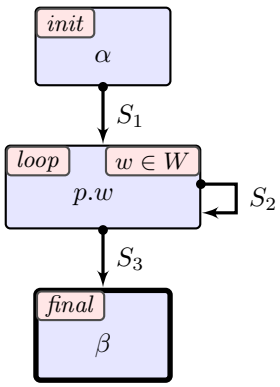

Figure 2. While diagram (While)

The soundness and completeness of these rules have been proved in [BP08, BP11] and they have been mechanically verified in the PVS theorem prover.

Definition 6. An invariant diagram (P, T) is *totally correct* if $\vdash P \{\!|\, T \,|\!\} P$.

3.3. Correctness of while diagrams

We will specialize the rule (3) to a *while diagram,* which has the simple structure of Figure 2.

Here we assume that $(W, <)$ is a well ordered set and we write $w \in W$ in the upper right corner of the $loop$ situation to indicate that the invariant of the situation $loop$ is in fact $\vee p = \bigvee_{w \in W} p.w$, even if we just write $p.w$ as the situation constraint. Let $I = \{init, loop, final\}$ be the set of situations of the diagram from Figure 2. The order $<$ on I is defined by $final < loop < init$, and the order $<$ on $I \times W$ is defined as the lexicographic order, i.e.

$$(i, w) < (j, v) \Leftrightarrow i < j \vee (i = j \wedge w < v)$$

The order relation on $I \times W$ is also a well order.

Let $\mathsf{While} : I \times I \to \mathsf{mtran}$ be the transition diagram corresponding to Figure 2, given by

$$\mathsf{While} = \mathsf{Magic}[(init, loop) \mapsto S_1, (loop, loop) \mapsto S_2, (loop, final) \mapsto S_3]$$

If we take

$$\begin{aligned} X.w.init &= \alpha \\ X.w.loop &= p.w \\ X.w.final &= \beta \end{aligned}$$

then the rule (3) becomes

$$\frac{\begin{array}{l} \models X.w.init \{\!|\, S_1 \,|\!\} X_{<(init,w)}.loop \\ \models X.w.loop \{\!|\, S_2 \,|\!\} X_{<(loop,w)}.loop \\ \models X.w.loop \{\!|\, S_3 \,|\!\} X_{<(loop,w)}.final \end{array}}{\vdash_s \bigvee X \{\!|\, \mathsf{While} \,|\!\} \bigvee X \cap \neg \mathsf{grd.While}}$$

because all the other transitions are magic. We have

- $X_{<(init,w)}.loop$
  $=$ {Definition}
  $\bigvee_{(loop,v)<(init,w)} X.v.loop$
  $=$ {Definition of $X.v.loop$}
  $\bigvee_{v \in W} p.v$
  $=$ {Definition of union}
  $\vee p$

- $X_{<(loop,w)}.loop$
  $=$ {Definition}
  $\bigvee_{(loop,v)<(loop,w)} X.v.loop$
  $=$ {Definition of $<$ on pairs}
  $\bigvee_{v<w} X.v.loop$
  $=$ {Definition of X}
  $\bigvee_{v<w} p.v$
  $=$ {Definition}
  $p_{<w}$

and similarly $X_{<(loop,w)}.final = \beta$. The rule for the while diagram is simplified to

$$\frac{\begin{array}{l}\models \alpha \,\{\!|\, S_1 \,|\!\}\, \vee p \\ \models p.w \,\{\!|\, S_2 \,|\!\}\, p_{<w} \\ \models p.w \,\{\!|\, S_3 \,|\!\}\, \beta\end{array}}{\vdash_s \bigvee X \,\{\!|\, \mathsf{While} \,|\!\}\, \bigvee X \cap \neg \mathsf{grd.While}}$$

where $p_{<w} = \bigvee_{v<w} p.v$. Because w is chosen such that it does not occur free in β and S_3, the statement $\models p.w \,\{\!|\, S_3 \,|\!\}\, \beta$ is equivalent to $\models \vee p \,\{\!|\, S_3 \,|\!\}\, \beta$. The rule for While becomes:

$$\frac{\begin{array}{l}\models \alpha \,\{\!|\, S_1 \,|\!\}\, \vee p \\ \models p.w \,\{\!|\, S_2 \,|\!\}\, p_{<w} \\ \models \vee p \,\{\!|\, S_3 \,|\!\}\, \beta\end{array}}{\vdash_s \bigvee X \,\{\!|\, \mathsf{While} \,|\!\}\, \bigvee X \cap \neg \mathsf{grd.While}} \tag{5}$$

In practical examples, the predicate $p.w$ from the diagram 2 is the conjunction of the situation invariant γ and a formula of the form $t = w$, where $w \in W$ and t is a variant term ranging over the set W. In the invariant diagram, we then state the variant t by writing $t \in W$ in the upper right corner of the *loop* situation (as exemplified in the factorial example). If $p.w = (\gamma \wedge t = w)$, then

$\vee p = \gamma$ and $p_{<w} = \gamma \wedge t < w$

therefore the proof obligations of the while diagram can be simplified further:

$$\frac{\begin{array}{l}\models \alpha \,\{\!|\, S_1 \,|\!\}\, \gamma \\ \models \gamma \wedge t = w \,\{\!|\, S_2 \,|\!\}\, \gamma \wedge t < w \\ \models \gamma \,\{\!|\, S_3 \,|\!\}\, \beta\end{array}}{\vdash_s \bigvee X \,\{\!|\, \mathsf{While} \,|\!\}\, \bigvee X \cap \neg \mathsf{grd.While}} \tag{6}$$

To apply this rule for the factorial example we set

$$\begin{aligned} P.init &= (n \geq 0) \\ P.loop &= (n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1) \\ P.final &= (n \geq 0 \wedge x = n!) \end{aligned}$$

The factorial example is strictly correct, $\vdash_s P \,\{\!|\, \mathsf{Factorial} \,|\!\}\, P \cap \neg \mathsf{grd.Factorial}$ if the following proof obligations are true:

$$\begin{array}{l} \models n \geq 0 \,\{\!|\, x,\, i := 1,\, 1 \,|\!\}\, n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1 \\ \models n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1 \wedge n - i + 1 = w \,\{\!|\, [i \leq n]\,;\, x,\, i := x \cdot i,\, i+1 \,|\!\} \\ \qquad n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1 \wedge n - i + 1 < w \\ \models n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1 \,\{\!|\, [i > n] \,|\!\}\, n \geq 0 \wedge x = n! \end{array}$$

3.4. Final situations and liveness

The execution of a diagrams can start from any situation, and it can end in any situation. In practice, we are mainly interested in diagrams in which the execution is guaranteed to terminate in some predetermined *final* situations. For example we want the factorial diagram presented earlier to always terminate in situation *final*. An *invariant diagram with final situations* is a tuple (P, T, J) where (P, T) is an invariant diagram, and $J \subseteq I$ is a non-empty set of final situations. Final situations are emphasized in the diagram by drawing them with a thicker border line, as already shown in the factorial example.

The diagram (P, T, J) is correct if (P, T) always terminates in a situation in J (we then say that the diagram is *live* for the final situations J). This is made more formal in the the following definition.

Definition 7. An invariant diagram with final situations (P, T, J) is *totally correct*, if

$$\vdash P \,\{\!|\, T \,|\!\}\, P^J \tag{7}$$

where for all $i \in I$

$$P^J.i = \begin{cases} P.i & \text{if } i \in J \\ \mathsf{false} & \text{otherwise} \end{cases}$$

To prove $\vdash P \{\!|\, T \,|\!\} P^J$ we have to find $X : W \to I \to \mathsf{pred}$ such that

$$\left(\forall i, j, w : \models X.w.i \{\!|\, T_{i,j} \,|\!\} X_{<(i,w)}.j\right),$$

$$P \subseteq \bigvee X,$$

and

$$\left(\bigvee X \cap \neg \mathsf{grd}.T \subseteq P^J\right)$$

are true. All these proof obligations are the same as in the case of a diagram without final states with the exception of the condition

$$\left(\forall i : (\bigvee X).i \cap (\neg \mathsf{grd}.T).i \subseteq P^J.i\right).$$

Instantiating it for a non-final situation $i \notin J$ yields

$$(\bigvee X).i \cap (\neg \mathsf{grd}.T).i \subseteq \mathsf{false}$$

which is equivalent to

$$(\bigvee X).i \subseteq (\mathsf{grd}.T).i$$

In a non-final situation i the invariant of i must imply the guard of i. This proof obligation guarantees that if the execution is in a non-final situation, then at least one transition is enabled.

In the factorial example, we require that execution can only terminate in the situation *final*. There are then two additional proof obligations, besides those we already identified above:

$$n \geq 0 \Rightarrow \mathsf{true}$$
$$n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1 \Rightarrow (i \leq n \vee i > n)$$

These proof obligations ensure that execution does not terminate in the situation $init$ nor in the situation $loop$.

4. Data refinement of invariant diagrams

Data refinement has proved to be a powerful tool in developing software systems. When writing a complex program, it is often useful to start with an abstract program on an abstract data structure, and gradually refine it to a more concrete program working on a concrete data structure. Using this approach the overall proof work is split into smaller tasks [Bac80a, BvW00, DE99].

Data refinement is often used to implement a data module with information hiding. The specification of the module defines the effect of access procedures in terms of abstract variables. The implementation of the module is in fact done in terms of concrete variables, in order to achieve efficiency. If we can prove data refinement for all access methods, then a user of the module will never see a difference, and may use the module and reason about its behavior as if it was really implemented in terms of the abstract variables.

The situation with invariant based programs is different. Here we are interested in deriving a concrete algorithm working on concrete variables. The abstraction is only useful if it saves us some verification effort, or can simplify the understanding and/or construction of the algorithm. It turns out that data refinement is in fact quite useful for this purpose also. In many cases, it is easier to first design an abstract program, working on some abstraction of the intended state, and prove that it satisfies our requirements, and then refine this to a more concrete program that works on the state space that we really want to (or have to) use.

4.1. Data refinement of transitions

We introduce first some general concepts about data refinement of statements, and we specialize them later for transitions of diagrams.

Definition 8. Let $S, S', D, D' \in \mathsf{mtran}$ be monotonic predicate transformers. The predicate transformer S is *data refined* by S' via the data abstraction statements D and D', denoted $S \sqsubseteq_{D,D'} S'$, if

$$D \,;\, S \sqsubseteq S' \,;\, D'$$

Data refinement satisfies some monotonicity properties which are summarized by the next theorem.

Theorem 9. (Data refinement properties).

1. $S \sqsubseteq_{D,D'} U \wedge S' \sqsubseteq_{D,D'} U' \Rightarrow S \sqcap S' \sqsubseteq_{D,D'} U \sqcap U'$

2. $S \sqsubseteq_{D,D'} U \wedge S \sqsubseteq_{D,D'} U' \Rightarrow S \sqsubseteq_{D,D'} U \sqcap U'$
3. $S \sqsubseteq_{D,D'} \mathsf{magic}$
4. $S \sqsubseteq S' \wedge S' \sqsubseteq_{D,D'} S'' \Rightarrow S \sqsubseteq_{D,D'} S''$
5. $p \subseteq q \Rightarrow \{\, p \,\} \sqsubseteq \{\, q \,\}$
6. $S \sqsubseteq S' \Rightarrow S \,;\, S'' \sqsubseteq S' \,;\, S''$

The next theorem is the main tool for proving the correctness of a statement S' that is a data refinement of another statement S.

Theorem 10. If $S, S', D, D' \in \mathsf{mtran}$ then

$$(\models p \,\{\!|\, S \,|\!\}\, q) \wedge (\{p\} \,;\, S \sqsubseteq_{D,D'} S') \;\Rightarrow \models D.p \,\{\!|\, S' \,|\!\}\, D'.q$$

Proof. We assume $(\models p \,\{\!|\, S \,|\!\}\, q) \Leftrightarrow (p \subseteq S.q)$ and $(\{p\} \,;\, S \sqsubseteq_{D,D'} S') \Leftrightarrow (D \,;\, \{\, p \,\} \,;\, S \sqsubseteq S' \,;\, D')$.

$\bullet$ $\models D.p \,\{\!|\, S' \,|\!\}\, D'.q$
$=$ {Definition}
$D.p \subseteq (S' \,;\, D').q$
$\Leftarrow$ {Assumptions}
$D.p \subseteq (D \,;\, \{\, p \,\} \,;\, S).q$
$=$ {Function composition and definitions}
$D.p \subseteq D.(p \wedge S.q)$
$=$ {Assumptions}
$D.p \subseteq D.p$
$=$ {Properties of set inclusion}
true

We use this theorem to establish the correctness of a concrete statement S', given that we have already shown that a corresponding more abstract statement is correct. Let us assume that we know (or have proved previously) that $\models p \,\{\!|\, S \,|\!\}\, q$ holds. Assume further that we can prove that S' is a data refinement of S, $\{p\} \,;\, S \sqsubseteq_{D,D'} S'$, when D and D' are chosen appropriately. From this we may then deduce directly that statement S' satisfies the correctness requirement $\models D.p \,\{\!|\, S' \,|\!\}\, D'.q$. Here $D.p$ gives us the concrete precondition and $D'.q$ the concrete post-condition that hold for S'. We are only interested in the correctness of statement S', but proving the correctness of S and the data refinement may be easier in many cases than trying to prove the correctness of S' directly.

The next theorem allows us to prove that a sequential composition of an assert statement and a demonic update is refined by a statement S, by reducing this to proving total correctness of S with respect to specific pre- and post-conditions.

Theorem 11. (Data refinement to Hoare). If $S, D \in \mathsf{mtran}$, $p \in \mathsf{pred}$, and $Q, R : \Sigma \to \mathsf{pred}$, then

$$(\{p\} \,;\, [Q] \sqsubseteq_{\{R\},D} S) \;=\; (\forall \sigma_0 : \models (\lambda \sigma : R.\sigma.\sigma_0 \wedge p.\sigma_0) \,\{\!|\, S \,|\!\}\, D.(Q.\sigma_0))$$

Proof.

$\bullet$ $\{\, p \,\} \,;\, [\, Q \,] \sqsubseteq_{\{\, R \,\},D} S$
$=$ {Definition of data refinement}
$\{\, R \,\} \,;\, \{\, p \,\} \,;\, [\, Q \,] \sqsubseteq S \,;\, D$
$=$ {Definition of $\sqsubseteq$}
$(\forall q, \sigma : (\{\, R \,\}.(\{\, p \,\}, ([\, Q \,].q))).\sigma \Rightarrow S.(D.q).\sigma)$
$=$ {Definition of angelic update, assert and demonic update}
$(\forall q, \sigma, \sigma_0 : R.\sigma.\sigma_0 \wedge p.\sigma_0 \wedge Q.\sigma_0 \subseteq q \Rightarrow S.(D.q).\sigma)$
$=$ {Monotonicity of S}
$(\forall \sigma, \sigma_0 : R.\sigma.\sigma_0 \wedge p.\sigma_0 \Rightarrow S.(D.(Q.\sigma_0)).\sigma)$
$=$ {Definition of $\subseteq$}
$(\forall \sigma_0 : (\lambda \sigma : R.\sigma.\sigma_0 \wedge p.\sigma_0) \subseteq S.(D.(Q.\sigma_0)))$
$=$ {Definition of valid Hoare triple}
$(\forall \sigma_0 : \models (\lambda \sigma : R.\sigma.\sigma_0 \wedge p.\sigma_0) \,\{\!|\, S \,|\!\}\, D.(Q.\sigma_0))$

This theorem will be used to generate proof obligations for data refinement statements.

The diagram from Figure 3 describes data refinement of a transition:

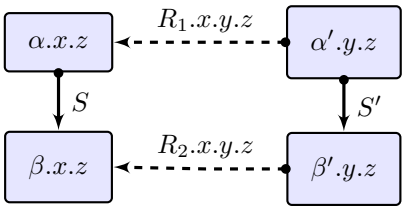

Figure 3. Data refinement of a transition

The *abstract* statement S modifies the global variables z and abstract variables x, and leads from an initial abstract situation $\alpha.x.z$ to a final abstract situation $\beta.x.z$. The *concrete* statement S' modifies the global variables z and concrete variables y and leads from an initial concrete situation $\alpha'.y.z$ to a final concrete situation $\beta'.y.z$. The *data abstraction relation* $R_1.x.y.z$ describes how the abstract variable x is related to the concrete variables y and z in the initial situation. Similarly for $R_2.x.y.z$ in the final situation.

We can define data abstraction in terms of an angelic statement D that computes for each concrete state some abstract state. Let us define

$$\begin{aligned} D_1 &= \{x := x' \mid R_1.x'.y.z \wedge \alpha'.y.z\} \\ D_2 &= \{x := x' \mid R_2.x'.y.z \wedge \beta'.y.z\} \end{aligned}$$

Here the abstraction (*decoding)* statements D_1 and D_2 include both the abstraction relation and the concrete invariant.

Let us now further assume that the abstract statement S is just a demonic assignment,

$$S = [\, x,\, z := x',\, z' \mid Q.x.z.x'.z' \,]$$

We want to refine the abstract statement in the context where the initial situation $\alpha.x.z$ holds. This is expressed by the data refinement

$$\{\alpha.x.z\}; S \sqsubseteq_{D_1,D_2} S'.$$

Theorem 12. Assume that D_1, D_2 and S are defined as above. The statement $\{\alpha.x.z\}; S \sqsubseteq_{D_1,D_2} S'$ is then equivalent to

$$(\forall x_0, z_0 :\models (z = z_0 \wedge R_1.x_0.y.z \wedge \alpha'.y.z \wedge \alpha.x_0.z_0) \,\{\!|\, S' \,|\!\}\, (\exists x : R_2.x.y.z \wedge \beta'.y.z \wedge Q.x_0.z_0.x.z))$$

Proof.

- $\{\, \alpha.x.z \,\}; S \sqsubseteq_{D_1,D_2} S'$

$=$ {Definitions}

$\{\, \alpha.x.z \,\}\,; [\, \lambda\sigma, \sigma' : Q.(x.\sigma).(z.\sigma).(x.\sigma').(z.\sigma') \,]$
$\sqsubseteq_{\{\, \lambda\sigma,\sigma' : z.\sigma = z.\sigma' \wedge R_1.(x.\sigma').(y.\sigma).(z.\sigma) \wedge \alpha'.(y.\sigma).(z.\sigma) \,\},D_2} S'$

$=$ {Theorem 11}

$(\forall \sigma_0 :\models (\lambda\sigma : z.\sigma = z.\sigma_0 \wedge R_1.(x.\sigma_0).(y.\sigma).(z.\sigma) \wedge \alpha'.(y.\sigma).(z.\sigma) \wedge \alpha.(x.\sigma_0).(z.\sigma_0)) \,\{\!|\, S' \,|\!\}$
$\{\, x := x' \mid R_2.x'.y.z \wedge \beta'.y.z \,\}.(Q.(x.\sigma_0).(z.\sigma_0).x.z))$

$=$ {State σ_0 has only two components $x.\sigma_0$ and $z.\sigma_0$}

$(\forall x_0, z_0 :\models (\lambda\sigma : z.\sigma = z_0 \wedge R_1.x_0.(y.\sigma).(z.\sigma) \wedge \alpha'.(y.\sigma).(z.\sigma) \wedge \alpha.x_0.z_0)) \,\{\!|\, S' \,|\!\}$
$\{\, x := x' \mid R_2.x'.y.z \wedge \beta'.y.z \,\}.(Q.x_0.z_0.x.z)$

$=$ {Definitions and Lemma 3}

$(\forall x_0, z_0 :\models (z = z_0 \wedge R_1.x_0.y.z \wedge \alpha'.y.z \wedge \alpha.x_0.z_0) \,\{\!|\, S' \,|\!\}$
$(\exists x : R_2.x.y.z \wedge \beta'.y.z \wedge Q.x_0.z_0.x.z)$

Consider the situation in the diagram from Figure 3. The initial situation has the constraint $\alpha'.y.z$ and the final situation has the constraint $\beta'.y.z$. Assume now that we want the concrete variables y and z to also satisfy the constraint $D_1.(\alpha.x.z)$, which says that there exists some value x such that $R_1.x.y.z \wedge \alpha.x.z$ holds. In other words, the variables y and z should also represent some abstract variables x that satisfy the abstract situation constraint. This means that the initial situation has the overall constraint

$$(\exists x : R_1.x.y.z \wedge \alpha'.y.z \wedge \alpha.x.z)$$

Similarly, we can argue that the final situation has the overall constraint

$$(\exists x : R_2.x.y.z \wedge \beta'.y.z \wedge \beta.x.z)$$

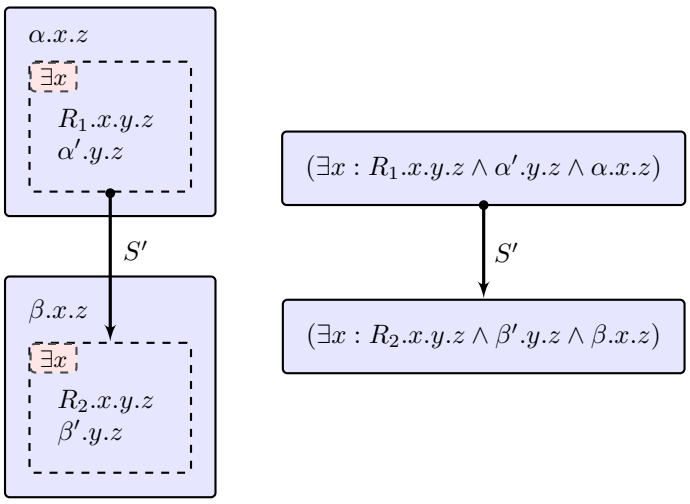

Figure 4. Nested vs. flattened diagram

This divides the constraint of the situation into two parts, a concrete and an abstract part. We are now looking for a concrete transition S' that leads from the initial situation to the final situation, when interpreted in this way. We describe this in an invariant diagram as shown on the left hand side of Figure 4.

The notation on the left indicates that the invariant of the nested situation is the conjunction of the constraints in the outer situation and the inner situation, but such that the variable x is removed by existentially quantifying it. The right hand diagram is equivalent to the left hand diagram, but written without the data abstraction notation. The advantage of the left hand side notation is that it shows the structure of the situation, how it is built up from a concrete and an abstract requirement.

The transition S' is now correct if

$$D_1.(\alpha.x.z) \,\{\![\, S' \,]\!\}\, D_2.(\beta.x.y) \tag{8}$$

or, writing it out explicitly, if

$$(\exists x : R_1.x.y.z \wedge \alpha'.y.z \wedge \alpha.x.z) \,\{|\, S' \,|\}\, (\exists x : R_2.x.y.z \wedge \beta'.y.z \wedge \beta.x.z)$$

holds. Based on Theorem 10, we can prove (8) in two steps:

(i) the abstract transition S is correct, and

(ii) the concrete transition S' is a data refinement of the abstract transition S.

The theorem is specialized for our purposes as follows:

Theorem 13.

$$\alpha.x.z \,\{\![\, S \,]\!\}\, \beta.x.z \qquad (i)$$

$\wedge$

$$(\forall x_0, z_0 : z = z_0 \wedge R_1.x_0.y.z \wedge \alpha'.y.z \wedge \alpha.x_0.z_0 \,\{\![\, S' \,]\!\}\, \beta'.y.z \wedge (\exists x : R_2.x.y.z \wedge Q.x_0.z_0.x.z)) \qquad (ii)$$

$\Rightarrow$

$$D_1.(\alpha.x.z) \,\{\![\, S' \,]\!\}\, D_2.(\beta.x.z)$$

Proof. This is a consequence of Theorem 10 and Theorem 12. □

Proving assumptions (i) and (ii) in the theorem will in fact establish the correctness of the diagram from Figure 5.

This shows both the abstract transition and the concrete transition, and explains how the concrete transition is derived from the abstract transition. The verification of both the abstract and the concrete transition can be done using only information that is explicitly given in the diagram.

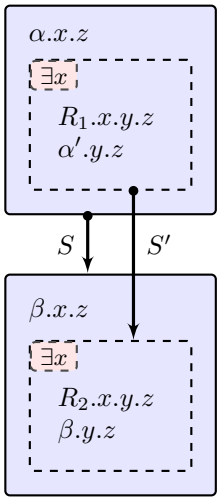

Figure 5. Abstract and concrete transitions

4.2. Data refinement of diagrams

The previous subsection has introduced data refinement for transitions, and it has introduced a diagrammatic notation for data refinement. It has also reduced the proof of correctness for the concrete transition to the proof of correctness for the abstract transition and the proof of data refinement between the abstract and concrete transitions. This subsection extends these concepts to diagrams.

A transition diagram T is data refined by another transition diagram T' via the data abstraction statements $D : I \to \mathsf{mtran}$ if every transition $T.i.j$ is data refined by $T'.i.j$. Formally we have:

Definition 14. (*Data refinement of transition diagrams*). The transition diagram T is refined by T' via the data abstraction statements $D : I \to \mathsf{mtran}$, denoted $T \sqsubseteq_D T'$ if

$$(\forall i, j : T.i.j \sqsubseteq_{D_i, D_j} T'.i.j)$$

In order to prove for transition diagrams a theorem similar to Theorem 10 we need to introduce a couple of new constructs.

The sequential composition of transition diagrams is defined similarly to the product of matrices, where the product is replaced by sequential composition and addition is replaced by the demonic choice.

Definition 15. (*Sequential composition of transition diagrams*) For transition diagrams $T, T' : I \to I \to \mathsf{mtran}$, the sequential composition of T and T', denoted $T\,;\,T' : I \to I \to \mathsf{mtran}$ is defined by

$$(T\,;\,T').i.j = \bigsqcap_{k \in I} T.i.k\,;\,T.k.j$$

Intuitively the transition between two situations i and j in the sequential composition $T\,;\,T'$ is the demonic choice over situation k of executing the transition $T.i.k$ from i to k in T followed by transition $T'.k.j$ from k to j in T'.

Definition 16. (*Assert transition diagram*) For an indexed predicate $P : I \to \mathsf{mtran}$ the assert transition diagram $\{\, P \,\} : I \to I \to \mathsf{mtran}$ is given by

$$\{\, P \,\}.i.j = \begin{cases} \{\, P.i \,\} & \text{if } i = j \\ \mathsf{magic} & \text{otherwise} \end{cases}$$

The assert diagram of an indexed predicate P has the transitions $\{\, P.i \,\}$ on the diagonal and it has magic (the neutral element for demonic choice) on the other transitions. The transitions of the sequential composition of $\{\, P \,\}$ and T are

$$(\{\, P \,\}\,;\,T).i.j = \{\, P.i \,\}\,;\,T.i.j$$

because $\mathsf{magic}\,;\,S = \mathsf{magic}$ and $\mathsf{magic} \sqcap S = S$.

If $D : I \to \mathsf{mtran}$ are indexed transitions and $P : I \to \mathsf{mtran}$ are indexed predicates, then $D.P : I \to \mathsf{pred}$ is the point-wise extension of the function application that we defined earlier:

$$(D.P).i = (D.i).(P.i)$$

Next theorem is the main theorem used in proving total correctness of a concrete diagram T' by proving the total correctness of a concrete diagram T and by proving that T is data refined by T'.

Theorem 17. (Data refinement for diagrams) Let $T, T' : I \to I \to \mathsf{mtran}$, $D : I \to \mathsf{mtran}$, and $P : I \to \mathsf{pred}$. If

$D.i$ are disjunctive then

$$(\vdash_s P \{\!|\, T \,|\!\} Q) \wedge (\{P\}\,;\, T \sqsubseteq_D T') \;\Rightarrow \vdash_s D.P \{\!|\, T' \,|\!\} D.P \cap \neg\mathsf{grd}.T'$$

Proof. Assume that $\vdash_s P \{\!|\, T \,|\!\} Q$, then there exists $X : W \to I \to \mathsf{pred}$ such that $P = \bigvee X$ and $Q = \bigvee X \cap \neg\mathsf{grd}.T$, and

$$\big(\forall i, j, w :\models X.w.i \{\!|\, T.i.j \,|\!\} X_{<(i,w)}.j\big)$$

Assume also $\{P\}\,;\, T \sqsubseteq_D T'$, then

$$(\forall i, j : \{\, P.i \,\}\,;\, T.i.j \sqsubseteq_{D_i,D_j} T'.i.j)$$

- $\vdash_s D.P \{\!|\, T' \,|\!\} D.P \cap \neg\mathsf{grd}.T'$

$\Leftrightarrow$ {Definition of $\vdash_s$}

$(\exists Y : (D.P = \bigvee Y) \wedge (D.P \cap \neg\mathsf{grd}.T' = \bigvee Y \cap \neg\mathsf{grd}.T')$
$\wedge(\forall i, j, w :\models Y.w.i \{\!|\, T'.i.j \,|\!\} Y_{<(i,w)}.j))$

$\Leftrightarrow$ {Simplifications}

$\big(\exists Y : (D.P = \bigvee Y) \wedge \big(\forall i, j, w :\models Y.w.i \{\!|\, T'.i.j \,|\!\} Y_{<(i,w)}.j\big)\big)$

$\Leftarrow$ {Existential quantifier elimination $Y.w.k = (D.k).(X.w.k)$}

$(D.P = \bigvee(\lambda v : \lambda k : (D.k).(X.v.k)))$
$\wedge \big(\forall i, j, w :\models (D.i).(X.w.i) \{\!|\, T'.i.j \,|\!\} (\lambda v : \lambda k : (D.k).(X.v.k))_{<(i,w)}.j\big)$

$\Leftrightarrow$ {Definition of $\bigvee$ for indexed predicates ($P \cup Q = (\lambda k : P.k \cup Q.k)$)}

$(D.P = (\lambda k : \bigvee(\lambda v : (D.k).(P.v.k))))$
$\wedge \big(\forall i, j, w :\models (D.i).(X.w.i) \{\!|\, T'.i.j \,|\!\} (\lambda k : (\lambda v : (D.k).(X.v.k))_{<(i,w)}).j\big)$

$\Leftrightarrow$ {$D.k$, $k \in I$ are disjunctive}

$(D.P = (\lambda k : (D.k).((\bigvee X).k))$
$\wedge \big(\forall i, j, w :\models (D.i).(P.w.i) \{\!|\, T'.i.j \,|\!\} (\lambda k : (D.k).(X_{<(i,w)}.k))).j\big)$

$\Leftrightarrow$ {Assumptions and definition of $\bigvee$}

$D.P = D.P \wedge \big(\forall i, j, w :\models (D.i).(P.w.i) \{\!|\, T'.i.j \,|\!\} (D.k).(X_{<(i,w)}.j)\big)$

$\Leftrightarrow$ {Simplifications}

$\big(\forall i, j, w :\models (D.i).(X.w.i) \{\!|\, T'.i.j \,|\!\} (D.k).(X_{<(i,w)}.j)\big)$

$\Leftarrow$ {Universal quantifier elimination}

$\models (D.i).(X.w.i) \{\!|\, T'.i.j \,|\!\} (D.k).(X_{<(i,w)}.j)$

$\Leftarrow$ {Theorem 10}

$\models (X.w.i \{\!|\, T.i.j \,|\!\} X_{<(i,w)}.j) \wedge \{\, X.w.i \,\}\,;\, T.i.j \sqsubseteq_{D_i,D_j} T'.i.j$

$\Leftrightarrow$ {Assumptions}

$\{\, X.w.i \,\}\,;\, T.i.j \sqsubseteq_{D_i,D_j} T'.i.j$

$\Leftarrow$ {Theorem 9.4, 5, 6}

$\{\, (\bigvee X).i \,\}\,;\, T.i.j \sqsubseteq_{D_i,D_j} T'.i.j$

$\Leftrightarrow$ {Assumptions}

$\{\, P.i \,\}\,;\, T.i.j \sqsubseteq_{D_i,D_j} T'.i.j$

$\Leftrightarrow$ {Assumptions}

true

4.3. Data refinement of the while diagram

Next we show how this theorem applies to the While diagram introduced earlier. Consider the diagram from Figure 6. We denote by X the indexed predicates given by $X.w.init = \alpha$, $X.w.loop = p.w$, and $X.w.final = \beta$, by P the indexed predicate given by $P.init = \alpha$, $P.loop = \vee p$, and $P.final = \beta$, and by D the indexed transition given by $D.init = \{\, x := x' \mid R_1.x.y.z \,\}$, $D.loop = \{\, x := x' \mid R_2.x.y.z \,\}$, and $D.final = \{\, x := x' \mid R_3.x.y.z \,\}$.

We are interested in proving correctness of the concrete diagram from Figure 7.

The strict correctness statement for this diagram can be stated as

$$\vdash_s D.P \{\!|\, \mathsf{ConcWhile} \,|\!\} D.P \cap \neg\mathsf{grd}.\mathsf{ConcWhile}$$

and using Theorem 17 it is reduced to the following statements:

$$\vdash_s P \{\!|\, \mathsf{While} \,|\!\} P \wedge \neg\mathsf{grd}.\mathsf{While}$$

$$\{\, P \,\}\,;\, \mathsf{While} \sqsubseteq_D \mathsf{ConcWhile}$$

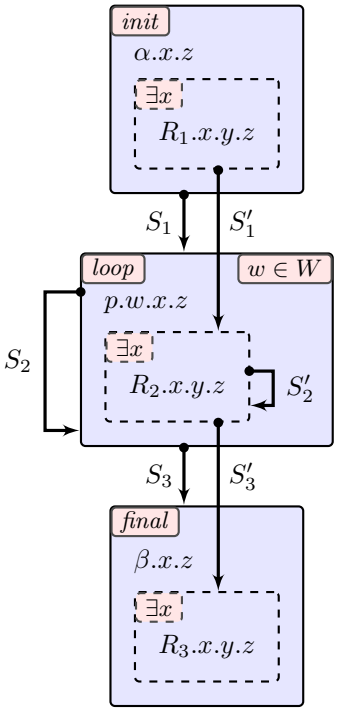

Figure 6. Data refinement of While

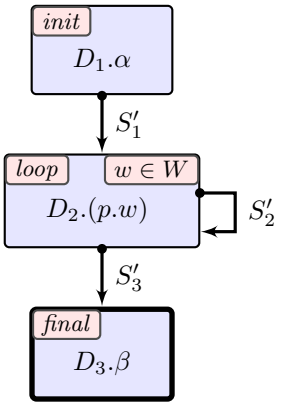

Figure 7. Concrete while diagram (ConcWhile)

Using the rule (5) we obtain the following proof obligations for the strict correctness of the abstract program:

$$\alpha \{\!|\, S_1 \,|\!\} \vee p$$
$$p.w \{\!|\, S_2 \,|\!\} p_{<w}$$
$$\vee p \{\!|\, S_3 \,|\!\} \beta$$

Assuming that $S_i = [x,\, z := x',\, z' : Q_i.x.z.x'.z']$, and using Theorem 12 we obtain the following proof obligations for the data refinement statement:

$$z = z_0 \wedge R_1.x_0.y.z \wedge \alpha.x_0.z_0 \{\!|\, S_1' \,|\!\} (\exists x : R_2.x.y.z \wedge Q_1.x_0.z_0.x.z)$$
$$z = z_0 \wedge R_2.x_0.y.z \wedge (\vee p).x_0.z_0 \{\!|\, S_2' \,|\!\} (\exists x : R_2.x.y.z \wedge Q_2.x_0.z_0.x.z)$$
$$z = z_0 \wedge R_2.x_0.y.z \wedge (\vee p).x_0.z_0 \{\!|\, S_3' \,|\!\} (\exists x : R_3.x.y.z \wedge Q_3.x_0.z_0.x.z)$$

If we prove that the abstract program terminates, then any data refinement of the abstract program will also terminate.

4.4. Liveness

From Theorem 17 it follows that termination for the abstract invariant diagram implies also termination of the concrete diagram. The same thing does not, however, hold for liveness. If we require that termination only happens in some specific final situations, then we need to prove this property explicitly for the concrete diagram. Termination in a specific final situation is thus not preserved by data refinement.

For example the factorial diagram (and any other diagram) is refined by Magic diagram, but Magic is not live. Termination occurs already in the $init$ situation.

If in the concrete while diagram we are interested in termination only in the $final$ situation we need to prove

$$\vdash Q \,\{\!|\, \mathsf{ConcWhile} \,|\!\}\, Q^J$$

where $Q = D.P$ and $J = \{final\}$ by proving

$$\begin{array}{l} \vdash_s Q \,\{\!|\, \mathsf{ConcWhile} \,|\!\}\, Q \cap \neg \mathsf{grd.ConcWhile} \\ Q \cap \neg \mathsf{grd.ConcWhile} \subseteq Q^J \end{array} \tag{9}$$

The second relation from (9) simplifies to:

$$Q.init \subseteq (\mathsf{grd.ConcWhile}).init$$

$$Q.loop \subseteq (\mathsf{grd.ConcWhile}).loop$$

that is, the invariants of $init$ and $loop$ situations must imply the guards of the transitions from $init$ and $loop$, respectively.

5. Data refinement of the DSW marking algorithm

We will now apply the data refinement technique presented above to construct the classical Deutsch-Schorr-Waite marking algorithm as an invariant based program. This algorithm marks all reachable nodes in an arbitrary directed graph. The algorithm is given for a graph structure represented using two pointers, left and right, associated with each node. A marking bit is associated with every node, and initially the marking bit is false for all nodes. An auxiliary bit called atom is also associated with every node (initially this bit can contain any value). The algorithm will mark exactly those nodes that are reachable from a given root node by a path on which all nodes have the atom bit false. Thus, y will not be marked if every path from the root to y contains a node with the atom bit true. If the algorithm should mark all reachable nodes, then the atom bit must initially be false for all nodes.

The algorithm changes the left and right pointers and the atom bit while performing the marking, but will restore the original values to these variables upon completion. The original algorithm has pairs of transitions for both left and right pointer functions. In order to simplify the algorithm we generalize it to solve the marking problem for a graph structure in which each node has a collection of pointers $lnk.k$ for $k \in \mathsf{label}$, instead of left and right only. This change enables us to treat uniformly the $lnk.k$ pointers. In the original algorithm we would need pairs of lemmas for both left and right pointer functions, however in the generalized version all these are replaced by single lemmas about the $lnk.k$ pointers.

We construct the classical DSW marking algorithm in four data refinement steps. First we build an algorithm that marks all nodes that can be reached from a special root node in an arbitrary directed graph. If node is a set of nodes (vertices), then the graph structure is given by a relation $\mathsf{next} \subseteq \mathsf{node} \times \mathsf{node}$. This initial algorithm uses an auxiliary variable X ranging over sets of nodes. The set X is initialized with the root element and, as long as X is nonempty, we either remove an element from X if all its successors are marked, or if there is an unmarked successor x of an element of X, then we mark x and add it to X. The algorithm finishes when the set X is empty. We prove that this algorithm marks all nodes reachable from root using the relation next. Also we prove termination for this initial algorithm.

The second algorithm uses a stack of nodes instead of a set. In the first version of the algorithm, any element of the set could be used to proceed with marking, the stack version always chooses the element at the top of the stack. If all successors of the top are marked, then the top is removed, otherwise an unmarked successor of the top of the stack is marked and pushed onto the stack. The stack stores the path from the current node to the root node. We derive the second algorithm with a data refinement from the first algorithm.

A second data refinement step replaces the relation next by the collection $lnk.k : \mathsf{node} \to \mathsf{node}$, $k \in \mathsf{label}$, of pointer functions and a function $lbl : \mathsf{node} \to \mathsf{label}$ which associate to every node a label from label. No extra variables are used for the stack, the stack is represented by temporarily changing the lnk and lbl variables. The data refinement shows that the new algorithm does the same computation as the previous algorithm, and that the variables lnk and lbl are restored to their initial values upon completion.

Finally, a third data refinement step replaces the general $lnk.k$ pointers with the left and right pointers, and thus yields the classical DSW algorithm. The function lbl is also replaced by the atom bit.

Here we present the steps of the construction of the algorithm, but we do not give any proofs. However, all proof obligations for this algorithm have been mechanically verified with the theorem prover Isabelle/HOL.

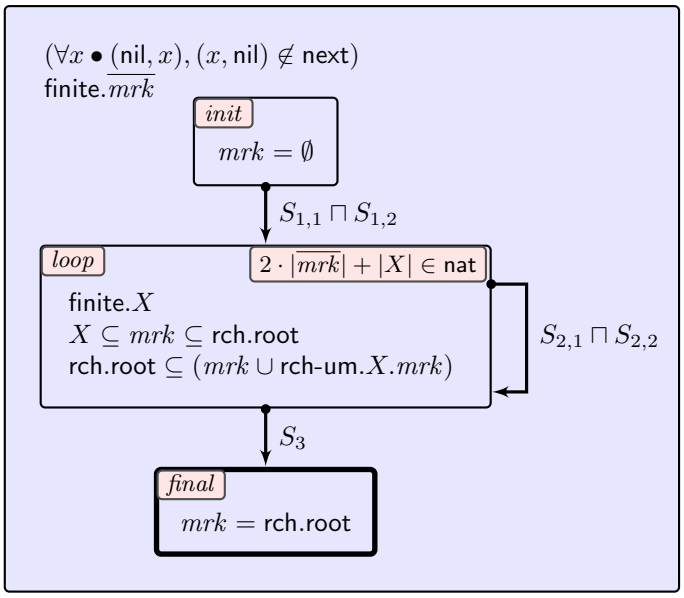

Figure 8. Marking using a set (SetMark)

5.1. Marking using a set

In the first algorithm we start with a set X containing the root element. As long as the set X is non-empty we repeat the following steps: if there exists an unmarked successor node x of a node in X, then we mark x and we add it to X; or if there is a node x in X such that all successors of x are marked, then we remove x from X. The algorithm terminates when the set X is empty.

Initially we know that the marking bit of all nodes is false, and we also assume that there is a finite number of nodes. The assumption that we have a finite number of nodes will be used to prove the termination of the algorithm. In the final state the marking bit of a node is true if and only if this node is reachable from root. While marking, we maintain the invariant that all nodes in the set X are marked, all nodes marked so far are reachable, and for every reachable node x either x is marked or there exists a path of un-marked nodes from a node in X to x. The termination is given by the fact that at each step, we either mark a node and add it to the set X, or we remove a node from X. Therefore, at each step, the term $2 \cdot |\overline{mrk}| + |X|$ is decreased, where mrk is the set of currently marked nodes and $|\overline{mrk}|$ stands for the number of unmarked nodes.

The algorithm works on the following program variables and constants (that together make up the abstract data structure):

$\mathsf{nil} : \mathsf{node}$

$\mathsf{root} : \mathsf{node}$

$\mathsf{next} \subseteq \mathsf{node} \times \mathsf{node}$

$mrk \subseteq \mathsf{node}$

$X \subseteq \mathsf{node}$

The type node is the type of all graph nodes. In practice this type is the type of all memory addresses (pointers). The constant nil is the null pointer, root is the initial graph node from which we compute the reachable nodes, and next is the relation which gives the graph structure. The set mrk is the variable in which we compute the set of reachable nodes. Initially mrk is set to empty-set, and on completion it will hold the set of all nodes reachable from root. The set X is an auxiliary variable which is initialized to the singleton root. We define the following auxiliary functions:

$$\begin{array}{lcl} \mathsf{rch.root} & := & \{x \mid (\mathsf{root}, x) \in \mathsf{next}^* \wedge x \neq \mathsf{nil}\} \\ \mathsf{rch\text{-}um}.X.mrk & := & \{x \mid (\exists y \in X : (y, x) \in \mathsf{next} \circ (\mathsf{next} \cap \overline{mrk} \times \overline{mrk})^*)\} \end{array}$$

where $\overline{mrk} = \mathsf{node} - mrk$ is the set complement of mrk, and for a relation R, R^* is the reflexive and transitive closure of R. The set rch.root contains all nodes reachable from root and $\mathsf{rch\text{-}um}.X.mrk$ contains all nodes reachable from X along unmarked nodes. The predicate $\mathsf{finite}.X$ is true if the set X is finite.

The invariant diagram from Figure 8 solves the marking problem in terms of the data representation above.

The statements in this diagram are defined by:

$$
\begin{array}{lcl}
S_{1,1} & := & [\,\mathsf{root} = \mathsf{nil}\,]\,;\; X := \emptyset \\
S_{1,2} & := & [\,\mathsf{root} \neq \mathsf{nil}\,]\,;\; X := \{\mathsf{root}\}\,;\; mrk := \{\mathsf{root}\}] \\
S_{2,1} & := & [X,\; mrk := X',\; mrk' \mid (\exists x \in X,\; y \notin mrk : (x,y) \in \mathsf{next} \\
 & & \qquad \wedge\, X' = X \cup \{y\} \wedge mrk' = mrk \cup \{y\})] \\
S_{2,2} & := & [X,\; mrk := X',\; mrk' \mid (\exists x \in X : (\forall y : (x,y) \in \mathsf{next} \Rightarrow y \in mrk) \\
 & & \quad \wedge\, X' = X - \{x\} \wedge mrk' = mrk)] \\
S_3 & := & [\,X = \emptyset\,]
\end{array}
$$

Theorem 18. The SetMark diagram is strictly correct:

$$\vdash_s \mathsf{Inv}\ \{\!|\ \mathsf{SetMark}\ |\!\}\ \mathsf{Inv} \cap \neg\mathsf{grd.SetMark}$$

where

$$
\begin{array}{lcl}
P.mrk & := & (\forall x : (\mathsf{nil}, x),\ (x, \mathsf{nil}) \notin \mathsf{next}) \wedge \mathsf{finite}.\overline{mrk} \\
\mathsf{Inv}.init.mrk & := & P.mrk \wedge mrk = \emptyset \\
\mathsf{Inv}.loop.X.mrk & := & P.mrk \wedge \mathsf{finite}.X \wedge X \subseteq mrk \subseteq \mathsf{rch.root} \\
 & & \qquad \wedge\, \mathsf{rch.root} \subseteq (mrk \cup \mathsf{rch\text{-}um}.X.mrk) \\
\mathsf{Inv}.final.mrk & := & P.mrk \wedge mrk = \mathsf{rch.root}
\end{array}
$$

Using the rule (3), the strict correctness of the SetMark diagram is reduced to the following proof obligations:

$$\mathsf{Inv}.init.mrk\ \{\!|\ S_{1,k}\ |\!\}\ \mathsf{Inv}.loop.X.mrk$$

$$(\mathsf{Inv}.loop.X.mrk \wedge 2 \cdot |\overline{mrk}| + |X| = w)\ \{\!|\ S_{2,k}\ |\!\}\ (\mathsf{Inv}.loop.X.mrk \wedge 2 \cdot |\overline{mrk}| + |X| < w)$$

$$\mathsf{Inv}.loop.X.mrk\ \{\!|\ S_3\ |\!\}\ \mathsf{Inv}.final.mrk$$

for all $k \in \{1, 2\}$.

5.2. Marking using a stack

Instead of X, we now start with a stack (list) S containing the root element. As long as the stack is non-empty we repeat the following step: If h is the top element of S and there exists an unmarked node y such that $(h, y) \in \mathsf{next}$, then we mark y and we add it to the top of the stack. Otherwise we pop the top element of the stack. The algorithm terminates when the stack is empty.

If S is a list with elements from node, then $\mathsf{hd}.S$ it the first (top) element of the list and $\mathsf{tl}.S$ is the list obtained from S by removing the first element. We call $\mathsf{hd}.S$ and $\mathsf{tl}.S$ the *head* and the *tail* of S, respectively. If S is empty, then $\mathsf{hd}.S$ is nil and $\mathsf{tl}.S$ is the empty list. We denote by $[\,]$ the empty list, $[x]$ the list with one element, and we use $+$ for list concatenation. The predicate $\mathsf{dist}.S$ is true if all elements of S are distinct of each other. The function set applied to S gives the set containing the elements of S.

The set variable X from the abstract program is replaced by a variable S which is a list of distinct elements. The data abstraction invariant states that the set X is equal to $\mathsf{set}.S$. All other variables from the abstract program are present in the concrete program with the same meaning.

The diagram for this refinement step is presented in Figure 9, where

$$
\begin{array}{lcl}
T_{1,1} & := & [\,\mathsf{root} = \mathsf{nil}\,]\,;\; S := [\,] \\
T_{1,2} & := & [\,\mathsf{root} \neq \mathsf{nil}\,]\,;\; S := [\mathsf{root}]\,;\; mrk := \{\mathsf{root}\} \\
T_{2,1} & := & [S,\; mrk := S',\; mrk' \mid S \neq [\,] \wedge (\exists y \notin mrk : (\mathsf{hd}.S, y) \in \mathsf{next} \\
 & & \qquad \wedge\, mrk' = mrk \cup \{y\} \wedge S' = [\,y\,] + S)] \\
T_{2,2} & := & [\,S \neq [\,] \wedge (\forall y : (\mathsf{hd}.S, y) \in \mathsf{next} \Rightarrow y \in mrk)]\,;\; S := \mathsf{tl}.S \\
T_3 & := & [\,S = [\,]\,]
\end{array}
$$

Theorem 19. The set diagram is data refined by the stack diagram:

$$\{\,\mathsf{Inv}\,\}\,;\ \mathsf{SetMark} \sqsubseteq_D \mathsf{StackMark}$$

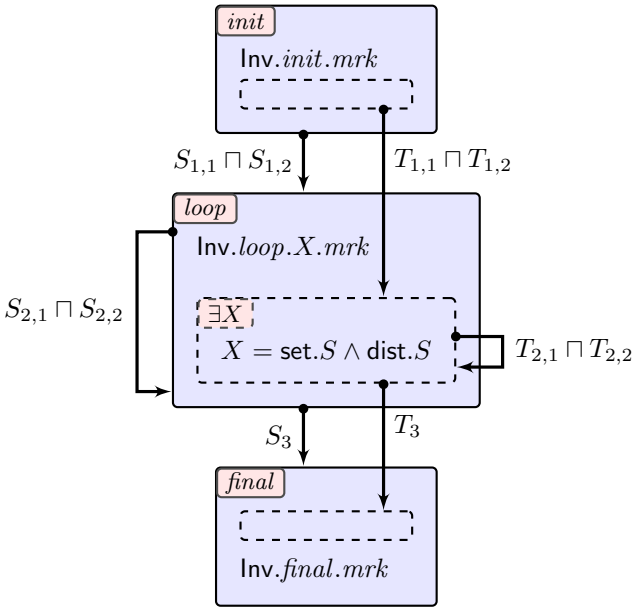

Figure 9. Marking using a stack (StackMark)

where

$$
\begin{array}{lcl}
D.init & = & \{\, \mathsf{true} \,\} \\
D.loop & = & \{\, X := X' \mid X' = \mathsf{set}.S \wedge \mathsf{dist}.S \,\} \\
D.final & = & \{\, \mathsf{true} \,\}
\end{array}
$$

Using Theorem 11, the proof obligations for this data refinement step, after some simplifications, are

$$
\begin{array}{l}
\mathsf{Inv}.init.mrk \wedge mrk = mrk_0 \,\{\!|\, T_{1,1} \,|\!\}\, \mathsf{root} = \mathsf{nil} \wedge \mathsf{dist}.S \wedge \mathsf{set}.S = \emptyset \\
\mathsf{Inv}.init.mrk \wedge mrk = mrk_0 \,\{\!|\, T_{1,2} \,|\!\}\, \mathsf{root} \neq \mathsf{nil} \wedge \mathsf{dist}.S \wedge \mathsf{set}.S = mrk = \{\mathsf{root}\} \\
\mathsf{dist}.S \wedge X_0 = \mathsf{set}.S \wedge \mathsf{Inv}.loop.X_0.mrk \wedge mrk = mrk_0 \,\{\!|\, T_{2,1} \,|\!\}\, \mathsf{dist}.S \wedge \\
\qquad (\exists x \in X_0, y \notin mrk_0 : (x,y) \in \mathsf{next} \wedge \mathsf{set}.S = X_0 \cup \{y\} \wedge mrk = mrk_0 \cup \{y\}) \\
\mathsf{dist}.S \wedge X_0 = \mathsf{set}.S \wedge \mathsf{Inv}.loop.X_0.mrk \wedge mrk = mrk_0 \,\{\!|\, T_{2,2} \,|\!\}\, \mathsf{dist}.S \wedge \\
\qquad (\exists x \in X_0 : (\forall y : (x,y) \in \mathsf{next} \Rightarrow y \in mrk_0) \wedge \mathsf{set}.S = X_0 - \{x\} \wedge mrk = mrk_0) \\
\mathsf{dist}.S \wedge X_0 = \mathsf{set}.S \wedge \mathsf{Inv}.loop.X_0.mrk \wedge mrk = mrk_0 \,\{\!|\, T_3 \,|\!\}\, X_0 = \emptyset
\end{array}
$$

Theorem 20. The stack diagram is strictly correct:

$$\vdash_s D.\mathsf{Inv}\ \{\!|\, \mathsf{StackMark} \,|\!\}\ (D.\mathsf{Inv} \cap \neg \mathsf{grd}.\mathsf{StackMark})$$

Proof. Using Theorems 17, 18, and 19.

5.3. Data refinement of the stack algorithm

In the third step the stack diagram is refined to a diagram where no extra memory is used. The variable next is replaced by two new variables lnk and lbl. The variable lnk is a collection of *pointer functions* $lnk.k : \mathsf{node} \to \mathsf{node}$ indexed by the set label. In general a function $f : \mathsf{node} \to \mathsf{node}$ is called a *pointer function*. For $x \in \mathsf{node}$, $lnk.k.x$ is the successor node of x along the function $lnk.k$. The variable $lbl : \mathsf{node} \to \mathsf{label}$ associates to each node a label. We call the structure given by the functions lnk and lbl a *pointer graph*. In a pointer graph a node x is reachable if there exists a path from the root to x along the links $lnk.k$ such that all nodes in this path are not nil and they are labeled by a special label none $\in$ label. Formally a node is reachable in a pointer graph if it is reachable from root using the relation $\mathsf{nxt}.lnk.lbl \subseteq \mathsf{node} \times \mathsf{node}$ given by

$$\mathsf{nxt}.lnk.lbl := \{(x,y) \mid (\exists k : lnk.k.x = y) \wedge x \neq \mathsf{nil} \wedge y \neq \mathsf{nil} \wedge lbl.x = \mathsf{none}\}$$

The first part of the loop data invariant in this step, denoted by α, is given by:

$$\alpha := (\mathsf{next} = \mathsf{nxt}.\mathsf{lnk}_0.\mathsf{lbl}_0)$$

where lnk_0 and lbl_0 are the initial values of the variables lnk and lbl.

The stack variable S is replaced by two new variables p and t ranging over nodes. Variable p stores the head of S, t stores the head of the tail of S, and the rest of S is stored by temporarily modifying the variables lnk and lbl. The predicate $\mathsf{stack}.lnk.lbl.t.S$ is true whenever S is a list of nodes, starting with t, such that if x is an element in S, then the next element in S is $lnk.(lbl.x).x$. That is, if we start from t and we apply successively the function $(\lambda x : lnk.(lbl.x).x)$ we obtain all elements of S in the order they occur in S. Formally the predicate stack is defined by induction on S:

$$\begin{array}{lcl}
\mathsf{stack}.lnk.lbl.x.[\,] & := & (x = \mathsf{nil}) \\
\mathsf{stack}.lnk.lbl.x.([y] + S) & := & (x \neq \mathsf{nil} \wedge x = y \wedge x \notin \mathsf{set}.S \wedge \mathsf{stack}.lnk.lbl.(lnk.(lbl.x).x).S)
\end{array}$$

While the algorithm is marking the graph nodes, the variables t points to a list T of nodes along the pointer function $(\lambda x : lnk.(lbl.x).x)$. The last element in T is the root node. The list T corresponds to the stack $\mathsf{tl}.S$ from the stack algorithm. The second part of the data invariant, denoted by β, is formally given by:

$$\beta := (p = \mathsf{hd}.S \wedge t = \mathsf{hd}.(\mathsf{tl}.S) \wedge \mathsf{nil} \notin \mathsf{set}.S \wedge \mathsf{stack}.lnk.lbl.t.(\mathsf{tl}.S))$$

The third part of the invariant is used to prove that, when the program terminates, the values of the variables lnk and lbl are restored to their initial values. For this purpose, two new functions are introduced $\mathsf{initlnk}.lnk.lbl.p.S : \mathsf{label} \to \mathsf{node} \to \mathsf{node}$ and $\mathsf{initlbl}.lbl.S : \mathsf{node} \to \mathsf{label}$. They compute the initial values of the program variables lnk and lbl based on their current values during marking and the value of the stack variable. These functions are defined by induction on the stack S:

$$\begin{array}{lcl}
\mathsf{initlnk}.lnk.lbl.p.[\,] & := & lnk \\
\mathsf{initlnk}.lnk.lbl.p.([x] + S) & := & \mathsf{initlnk}.(lnk[lbl.x,\, x \mapsto p)]).lbl.S \\
\mathsf{initlbl}.lbl.[\,] & := & lbl \\
\mathsf{initlbl}.lbl.([x] + S) & := & \mathsf{initlbl}.(lbl[x \mapsto \mathsf{none}]).S
\end{array}$$

The third component of the invariant, denoted by γ, is

$$\gamma := (\mathsf{lnk}_0 = \mathsf{initlnk}.lnk.lbl.p.(\mathsf{tl}.S) \wedge \mathsf{lbl}_0 = \mathsf{initlbl}.lbl.(\mathsf{tl}.S))$$

The invariant of this refinement step can be stated now as:

$$R.S.p.t.lnk.lbl := \alpha \wedge \beta \wedge \gamma$$

The variable p stores the current node, and at each step the algorithm checks if there is a successor x of p that is not marked (there exists a k such that $x = \mathsf{lnk}.k.p \notin mrk \cup \{\mathsf{nil}\}$ and $\mathsf{lbl}.p = \mathsf{none}$). Then x is marked, p is put into the stack ($lnk.k.p$ is changed to t, and $lbl.p$ is set to k), t is set to p, and x becomes the current node (p is set to x). If there is no such element, then the top element is removed from the stack: $lnk.(lbl.t).t$ is set to p, $lbl.t$ is set to none, p is set to t, and t is set to $lnk.(lbl.t).t$. The condition that a successor node of p is not marked is given by the following predicate:

$$\mathsf{g}.mrk.lbl.lnk.k.p := (lnk.k.p \notin mrk \cup \{\mathsf{nil}\} \wedge lbl.p = \mathsf{none})$$

The invariant linking the initial states and the final states of the abstract and concrete program is simpler. It only states that the variables lnk and lbl are equal to their initial values lnk_0 and lbl_0 respectively, as well as the graph given by next is equal to the graph defined by the pointer graph lnk, lbl.

The diagram for this second refinement step is now as shown in Figure 10, where

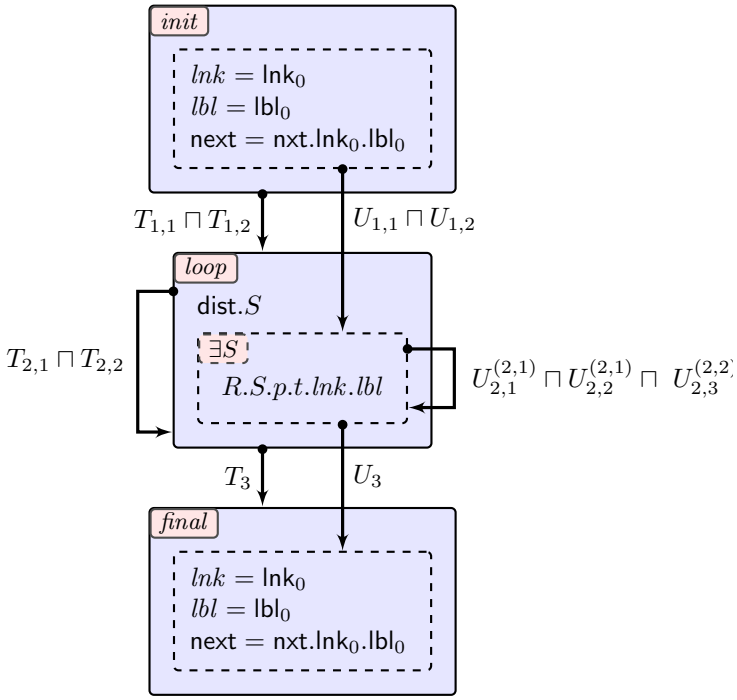

Figure 10. Marking with pointer functions (PtrMark)

$$
\begin{aligned}
U_{1,1} &:= [\,\mathsf{root} = \mathsf{nil}\,];\ p, t := \mathsf{nil}, \mathsf{nil} \\
U_{1,2} &:= [\,\mathsf{root} \neq \mathsf{nil}\,];\ p, t, mrk := \mathsf{root}, \mathsf{nil}, \{\mathsf{root}\} \\
U_{2,1} &:= [lnk, lbl, mrk, p, t := lnk', lbl', mrk', p', t' \mid p \neq \mathsf{nil} \\
&\quad \wedge (\exists k : \mathsf{g}.mrk.lbl.lnk.k.p \\
&\quad\quad \wedge mrk' = mrk \cup \{lnk.k.p\} \\
&\quad\quad \wedge lnk' = lnk[k, p \mapsto t] \\
&\quad\quad \wedge lbl' = lbl[p \mapsto k] \\
&\quad\quad \wedge p' = lnk.k.p \wedge t' = p)\ ] \\
U_{2,2} &:= [\,p \neq \mathsf{nil} \wedge (\forall k : \neg \mathsf{g}.mrk.lbl.lnk.k.p) \wedge t \neq \mathsf{nil}\,]; \\
&\quad lnk, lbl, p, t := lnk[(lbl.t), t \mapsto p], lbl[t \mapsto \mathsf{none}], t, lnk.(lbl.t).t \\
U_{2,3} &:= [\,p \neq \mathsf{nil} \wedge (\forall k : \neg \mathsf{g}.mrk.lbl.lnk.k.p) \wedge t = \mathsf{nil}\,];\ p := \mathsf{nil} \\
U_3 &:= [\,p = \mathsf{nil}\,]
\end{aligned}
$$

In Figure 10, the notation $U_{2,1}^{(2,1)}$, $U_{2,2}^{(2,1)}$, and $U_{2,3}^{(2,2)}$ is used to specify that $U_{2,1}$ refines $T_{2,1}$, $U_{2,2}$ refines $T_{2,1}$ and $U_{2,3}$ refines $T_{2,2}$. The data refinement of the diagram from Figure 10 holds if we prove that the transitions T are refined by the transitions U.

Let Q denote the following predicate:

$$Q.lnk.lbl := (lnk = \mathsf{lnk}_0 \wedge lbl = \mathsf{lbl}_0 \wedge \mathsf{next} = \mathsf{nxt}.\mathsf{lnk}_0.\mathsf{lbl}_0)$$

and

$$
\begin{aligned}
D'.init &= \{\, Q.lnk.lbl \,\} \\
D'.loop &= \{\, S := S' \mid R.S'.p.t.lnk.lbl \,\} \\
D'.final &= \{\, Q.lnk.lbl \,\}
\end{aligned}
$$

Theorem 21. The stack diagrams is data refined by the pointer diagram:

$$\{\, D.\mathsf{Inv} \,\};\ \mathsf{StackMark} \sqsubseteq_{D'} \mathsf{PtrMark}$$

Using Theorem 11, the proof obligations for this data refinement step, after some simplifications, are

$$Q.lnk.lbl \wedge mrk = mrk_0 \,\{\!|\, U_{1,1} \,|\!\}\, R.[\,].p.t.lnk.lbl$$
$$Q.lnk.lbl \wedge mrk = mrk_0 \,\{\!|\, U_{1,2} \,|\!\}\, R.[\mathsf{root}].p.t.lnk.lbl \wedge mrk = \{\mathsf{root}\}$$
$$R.S_0.p.t.lnk.lbl \wedge \mathsf{dist}.S_0 \wedge mrk = mrk_0 \,\{\!|\, U_{2,1} \,|\!\}\, (\exists y \notin mrk_0 : S_0 \neq [\,]$$
$$\wedge\, (\mathsf{hd}.S_0, y) \in \mathsf{next} \wedge mrk = mrk_0 \cup \{y\} \wedge R.([y] + S_0).p.t.lnk.lbl)$$
$$R.S_0.p.t.lnk.lbl \wedge \mathsf{dist}.S_0 \wedge mrk = mrk_0 \,\{\!|\, U_{2,2} \,|\!\}$$
$$R.(\mathsf{tl}.S_0).p.t.lnk.lbl \wedge S_0 \neq [\,] \wedge (\forall y : (\mathsf{hd}.S_0, y) \in \mathsf{next} \Rightarrow y \in mrk_0)$$
$$R.S_0.p.t.lnk.lbl \wedge \mathsf{dist}.S_0 \wedge mrk = mrk_0 \,\{\!|\, U_{2,3} \,|\!\}$$
$$R.(\mathsf{tl}.S_0).p.t.lnk.lbl \wedge S_0 \neq [\,] \wedge (\forall y : (\mathsf{hd}.S_0, y) \in \mathsf{next} \Rightarrow y \in mrk_0)$$
$$R.S_0.p.t.lnk.lbl \wedge \mathsf{dist}.S_0 \wedge mrk = mrk_0 \,\{\!|\, U_3 \,|\!\}\, Q.lnk.lbl \wedge S_0 = [\,]$$

In the above proof obligations the assumptions given by the Inv were omitted because they are not relevant in proving this refinement step. They could be omitted based on the following lemma

Lemma 22. If $p \in \mathsf{pred}$ and $R : \Sigma \to \mathsf{pred}$, then

$$(\forall s' :\models (\lambda s.R.s.s') \,\{\!|\, S \,|\!\}\, q) \;\Rightarrow\; \models (\{\, R \,\}.p) \,\{\!|\, S \,|\!\}\, q.$$

Theorem 23. The pointer diagram is strictly correct:

$$\vdash_s D'.(D.\mathsf{Inv}) \,\{\!|\, \mathsf{PtrMark} \,|\!\}\, D'.(D.\mathsf{Inv}) \cap \neg \mathsf{grd}.\mathsf{PtrMark}$$

Proof. Using Theorems 17, 18, 19, and 9.

5.4. Classical DSW marking algorithm

Finally, we construct the DSW marking algorithm by assuming that there are only two pointers (lft, rgt) from every node. The data invariant of this refinement step requires that label has exactly two distinct elements none and some, $lft = lnk.\mathsf{none}$, $rgt = lnk.\mathsf{some}$, and $at.x$ is true if and only if $lbl.x = \mathsf{some}$. Formally we have:

$$\begin{array}{lcl}
\mathsf{lnk}.lft.rgt.k & := & \begin{cases} lft & \text{if } k = \mathsf{none} \\ rgt & \text{if } k = \mathsf{some} \end{cases} \\
\mathsf{lbl}.at.x & := & \begin{cases} \mathsf{some} & \text{if } at.x \\ \mathsf{none} & \text{if } \neg at.x \end{cases} \\
R'.lnk.lbl.lft.rgt.at & := & (\mathsf{label} = \{\mathsf{none},\ \mathsf{some}\} \wedge lnk = \mathsf{lnk}.lft.rgt \wedge lbl = \mathsf{lbl}.at)
\end{array}$$

In this last data refinement step we need to introduce separate transitions for the left and right pointers. In the diagram below, the transition $U_{2,1}$ is refined by $V_{2,1}$ and $V_{2,2}$, the transition $U_{2,2}$ is refined by $V_{2,3}$ and $V_{2,4}$, and the transition $U_{2,3}$ is refined by $V_{2,5}$. If we would work with a left right pointer structure from the beginning we would need to introduce separate transitions for the left and right pointers already in the initial algorithm. Also many lemmas about $lnk.k$ pointers would need to have two versions for left and right.

The final data refinement step is described by the diagram from Figure 11, where

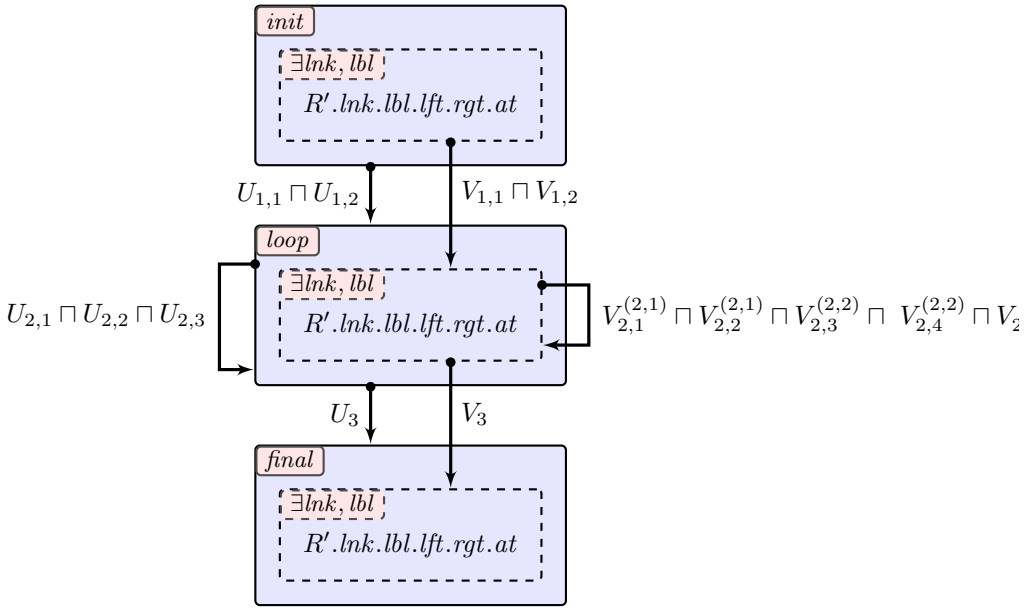

Figure 11. Classical marking algorithm (Mark)

$$
\begin{array}{lcl}
V_{1,1} & := & [\,\mathsf{root} = \mathsf{nil}\,];\ p,\ t := \mathsf{nil},\ \mathsf{nil} \\
V_{1,2} & := & [\,\mathsf{root} \neq \mathsf{nil}\,];\ p,\ t,\ mrk := \mathsf{root},\ \mathsf{nil},\ mrk[\mathsf{root} \mapsto \mathsf{true}] \\
V_{2,1} & := & [\,p \neq \mathsf{nil} \wedge \mathsf{h}.mrk.(at.p).(lft.p)\,];\ p,\ t,\ mrk,\ lft := \\
 & & \quad lft.p,\ p,\ mrk \cup \{lft.p\},\ lft[p \mapsto t] \\
V_{2,2} & := & [\,p \neq \mathsf{nil} \wedge \mathsf{h}.mrk.(at.p).(rgt.p)\,];\ p,\ t,\ mrk,\ rgt,\ at := \\
 & & \quad rgt.p,\ p,\ mrk \cup \{rgt.p\},\ rgt[p \mapsto t],\ at[p \mapsto \mathsf{true}] \\
V_{2,3} & := & [\,p \neq \mathsf{nil} \wedge \neg\mathsf{h}.mrk.(at.p).(lft.p) \wedge \neg\mathsf{h}.mrk.(at.p).(rgt.p) \wedge t \neq \mathsf{nil} \wedge \neg at.p\,]; \\
 & & \quad p,\ t,\ lft := t,\ lft.t,\ lft[t \mapsto p] \\
V_{2,4} & := & [\,p \neq \mathsf{nil} \wedge \neg\mathsf{h}.mrk.(at.p).(lft.p) \wedge \neg\mathsf{h}.mrk.(at.p).(rgt.p) \wedge t \neq \mathsf{nil} \wedge at.p\,]; \\
 & & \quad p,\ t,\ rgt,\ at := t,\ rgt.t,\ rgt[t \mapsto p],\ at[t \mapsto \mathsf{false}] \\
V_{2,5} & := & [\,p \neq \mathsf{nil} \wedge \neg\mathsf{h}.mrk.(at.p).(lft.p) \wedge \neg\mathsf{h}.mrk.(at.p).(rgt.p) \wedge t = \mathsf{nil}\,]; \\
 & & \quad p := \mathsf{nil} \\
V_3 & := & [\,p = \mathsf{nil}\,]
\end{array}
$$

and

$$\mathsf{h}.mrk.atom.x := x \notin mrk \cup \{\mathsf{nil}\} \wedge \neg atom$$

Let D'' be the diagram data refinement statement given by

$$D''.i = \{\ lnk, lbl = lnk', lbl' \mid R'.lnk'.lbl'.lft.rgt.at\ \}$$

for all $i \in I$.

Theorem 24. The pointer diagram is data refined by the DSW diagram:

$$\{\ D'.(D.\mathsf{Inv})\ \}\,;\ \mathsf{PtrMark} \sqsubseteq_{D''} \mathsf{Mark}$$

The proof obligations for this refinement step are tedious, but the refinement step is mainly a renaming of some variables, and these proof obligations were discharged almost automatically in the Isabelle theorem prover.

Theorem 25. The DSW diagram is strictly correct:

$$\vdash_s D''.(D'.(D.\mathsf{Inv}))\ \{\!|\ \mathsf{Mark}\ |\!\}\ D''.(D'.(D.\mathsf{Inv})) \cap \neg\mathsf{grd}.\mathsf{Mark}$$

Proof. Using Theorems 17, 18, 19, and 9.

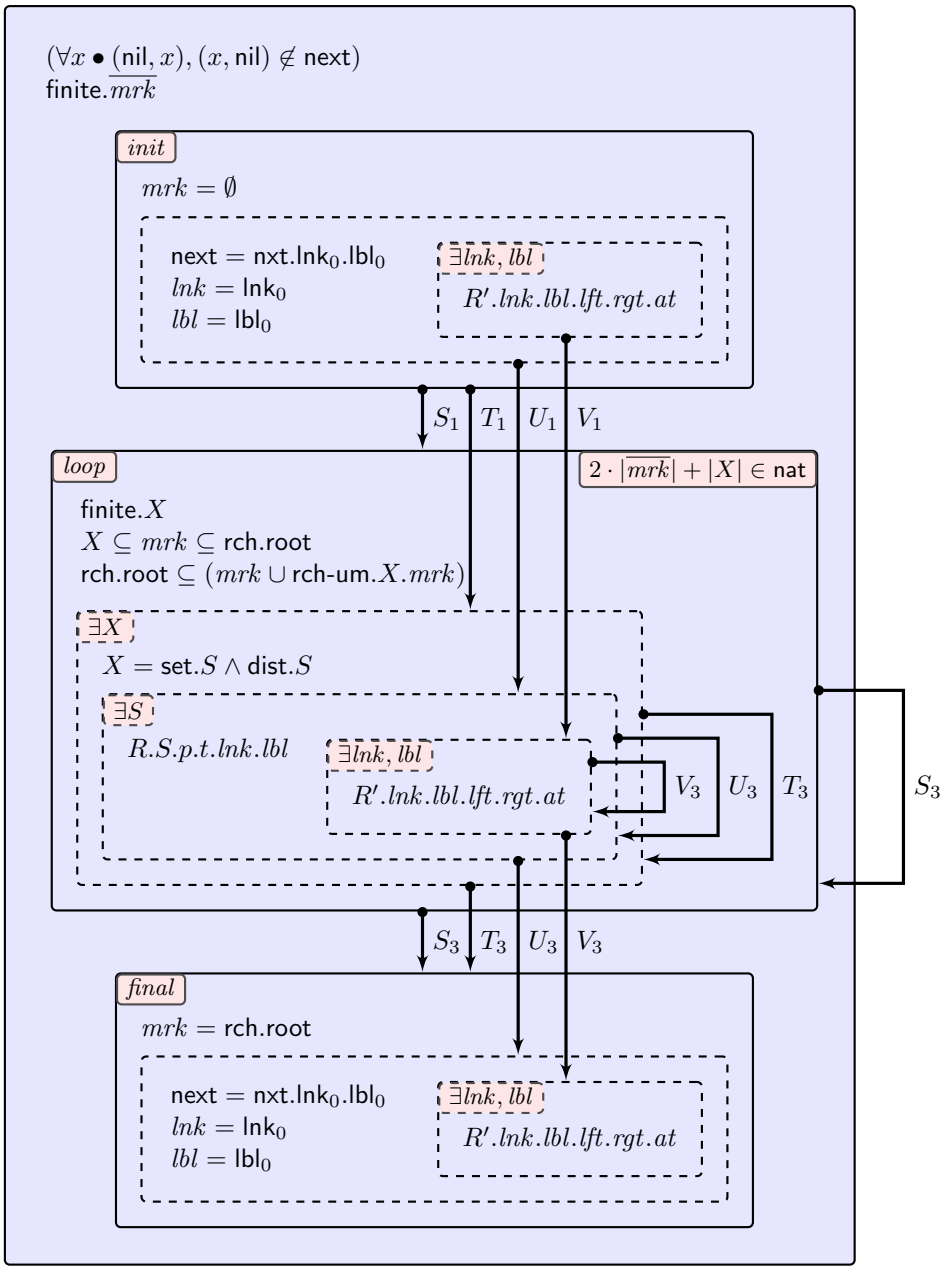

Figure 12. Complete diagram

5.5. Complete derivation

The diagram from Figure 12 collects all the refinement steps into a single diagram. The diagram shows explicitly all situations, and how they are built up as successive layers of abstraction. The transitions describe all successive versions of the marking algorithm. The diagram contains all the information that we need in order to verify that each version of the algorithm is correct.

5.6. Final algorithm

The main variables used in the final concrete program are lft, rgt, at, and mrk. The variables lft and rgt store the left and right successors of a graph node. The variable at defines what nodes are marked and is used during marking. The variable mrk contains at the end of the algorithm the set of all reachable nodes. The variables lft, rgt, and at are altered during marking, but they are restored to their initial values when the program terminates. The program uses also two auxiliary variables $p,\ t : \mathsf{node}$. Variable p stores the head of the stack S, and t stores the

head of the tail of S. The rest of S is stored by temporarily modifying the variables lft, rgt and at. Three auxiliary variables, lft_0, rgt_0, and $\mathsf{at}_0$are also used to record the initial values of lft, rgt, and at, respectively.

In this final program we are interested only in the initial and the final situations. These predicates should specify the marking problem as well as the fact that the initial values of the variables lft, rgt, and at are restored to their initial values when the program terminates. If the pre-condition and the post-condition of the diagram are given by

$$\begin{aligned}
\mathsf{Pre}.init &:= (at = \mathsf{at}_0 \wedge lft = \mathsf{lft}_0 \wedge rgt = \mathsf{rgt}_0 \wedge \mathsf{finite}.\overline{mrk} \wedge mrk = \emptyset)\\
\mathsf{Pre}.loop &:= \mathsf{false}\\
\mathsf{Pre}.final &:= \mathsf{false}
\end{aligned}$$

and

$$\begin{aligned}
\mathsf{Post}.init &:= \mathsf{false}\\
\mathsf{Post}.loop &:= \mathsf{false}\\
\mathsf{Post}.final &:= (at = \mathsf{at}_0 \wedge lft = \mathsf{lft}_0 \wedge rgt = \mathsf{rgt}_0 \wedge mrk = \mathsf{rch.root})
\end{aligned}$$

then the final correctness statement for the marking diagram is stated in the next theorem.

Theorem 26. The diagram Mark is correct with respect to Pre and Post predicates:

$\vdash \mathsf{Pre}\ \{|\ \mathsf{Mark}\ |\}\ \mathsf{Post}$

Proof. The result of Theorem 25 was obtained for arbitrary functions lnk_0, lbl_0, and next. Here we assume that these functions are given by the following definitions:

$$\begin{aligned}
\mathsf{lnk}_0 &:= \mathsf{lnk}.\mathsf{lft}_0.\mathsf{rgt}_0\\
\mathsf{lbl}_0 &:= \mathsf{lbl}.\mathsf{at}_0\\
\mathsf{next} &:= \mathsf{nxt}.\mathsf{lnk}_0.\mathsf{lbl}_0
\end{aligned}$$

Under these assumptions we can prove

$$\mathsf{Pre} \subseteq D''.(D'.(D.\mathsf{Inv})) \quad \text{and} \quad D''.(D'.(D.\mathsf{Inv})) \cap \neg\mathsf{grd.Mark} \subseteq \mathsf{Post}$$

Using Theorem 25 and the definition of correctness for diagrams it follows:

$\vdash \mathsf{Pre}\ \{|\ \mathsf{Mark}\ |\}\ \mathsf{Post}$

All results presented in this paper have been mechanically verified in the Isabelle/HOL theorem prover. We have used a number of features of the Isabelle theorem prover and we have benefited from a well developed library of reusable theories including the theory of complete lattices and well founded relations which are the bases of our development. We have used type variables for all types in our development. Using this mechanism we were able to replace the collection lnk of pointer functions by the lft and rgt functions by instantiating the type variable label with a type containing two elements. Further developments of our final algorithm are possible by instantiating the graph nodes with a concrete type of addresses. Another important feature used in the Isabelle implementation is the parametrization (locale) mechanism. In Isabelle a *locale* is a mathematical theory which can be parametrized by constants and type variables and it may contain axioms. A locale can be instantiated for concrete types and constants and the axioms must be proved for the concrete instantiations. We used this mechanism for defining the graph structure. Initially we have a locale introducing the nil element and the next relation as constants and satisfying the axiom that no element is related to nil by next or next^{-1}. This locale is instantiated by a more concrete structure where next is defined in terms of lnk_0 and lbl_0. The axiom was proved to hold for the instantiation. In the final refinement step we defined the lnk_0, lbl_0 functions in terms of lft_0, rgt_0, and at_0. This formalization is available online in the Archive of Formal Proofs [PB10a, PB10b].

6. Conclusions

In this paper we have shown how to carry out data refinement of invariant based programs, and have applied the technique to the construction of the classical Deutsch-Schorr-Waite graph marking algorithm. We have used predicate transformer semantics for the transitions in the invariant diagrams, and we have shown that termination is preserved by data refinement.

We have shown how the overall proof effort of the marking algorithm, is simplified by first proving a generalized version of it, and then data refining it to the classical version. All results presented in this paper have been proved mechanically using the Isabelle/HOL interactive theorem prover. This gives a very solid foundation of our results. Currently we are working on including data refinement into the Socos environment [BEM06, BEM07], which is specifically designed to support the construction of invariant based programs and to prove their correctness.

References

[Abr03] J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.

[Bac80a] R. J. Back. *Correctness preserving program refinements: proof theory and applications*, volume 131 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1980.

[Bac80b] R. J. Back. Semantic correctness of invariant based programs. In *International Workshop on Program Construction*, Chateau de Bonas, France, 1980.

[Bac83] R. J. Back. Invariant based programs and their correctness. In W. Biermann, G Guiho, and Y Kodratoff, editors, *Automatic Program Construction Techniques*, pages 223–242. MacMillan Publishing Company, 1983.

[Bac08] R. J. Back. Invariant based programming: Basic approach and teaching experience. *Formal Aspects of Computing*, 2008.

[BEM06] R. J. Back, J. Eriksson, and M. Myreen. Verifying invariant based programs in the SOCOS environment. In P. Boca, J. P. Bowen, and D. A. Duce, editors, *Teaching Formal Methods: Practice and Experience*, Electronic Workshops in Computing (eWiC). BCS, Dec 2006.

[BEM07] R. J. Back, J. Eriksson, and M. Myreen. Testing and verifying invariant based programs in the SOCOS environment. In *The International Conference on Tests And Proofs (TAP)*, 2007.

[BP08] R. J. Back and V. Preoteasa. Semantics and proof rules of invariant based programs. Technical Report 903, TUCS, Jul 2008.

[BP11] R.-J. Back and V. Preoteasa. Semantics and proof rules of invariant based programs. In *Proc. of 26th Symposium On Applied Computing - Software Verification and Testing Track*. ACM, March 2011.

[BvW98] R. J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.

[BvW00] R. J. Back and J. von Wright. Encoding, decoding and data refinement. *Formal Aspects of Computing*, 12:313–349, 2000.

[DE99] W. DeRoever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1999.

[Dij76] E.W. Dijkstra. *A discipline of programming*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976. With a foreword by C. A. R. Hoare, Prentice-Hall Series in Automatic Computation.

[Heh79] E. C. R. Hehner. do considered od: A contribution to the programming calculus. *Acta Informatica*, 11(4):287–304, 1979.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4), December 1972.

[Knu97] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[MN05] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[PB09] V. Preoteasa and R.-J. Back. Data refinement of invariant based programs. *Electronic Notes in Theoretical Computer Science*, 259:143 – 163, 2009. Proceedings of the 14th BCS-FACS Refinement Workshop (REFINE 2009).

[PB10a] V. Preoteasa and R.-J. Back. Semantics and data refinement of invariant based programs. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. http://afp.sourceforge.net/entries/DataRefinementIBP.shtml, May 2010. Formal proof development.

[PB10b] V. Preoteasa and R.-J. Back. Verification of the Deutsch-Schorr-Waite graph marking algorithm using data refinement. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. http://afp.sourceforge.net/entries/GraphMarkingIBP.shtml, May 2010. Formal proof development.

[Pnu05] A. Pnueli. Verification of procedural programs. In *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two*, pages 543–590, 2005.

[Rey78] J. C. Reynolds. Programming with transition diagrams. In D. Gries, editor, *Programming Methodology*. Springer Verlag, Berlin, 1978.

[SW67] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

[VE79] M. H. Van Emden. Programming with verification conditions. *IEEE Trans. Softw. Eng.*, 5(2):148–159, 1979.