A DSL for Integer Range Reasoning: Partition, Interval and Mapping Diagrams

Johannes Eriksson[1(✉)] and Masoumeh Parsa[2]

[1] Vaadin Ltd., Turku, Finland
joheriks@vaadin.com
[2] Department of Information Technologies, Åbo Akademi University, Turku, Finland
mparsa@abo.fi

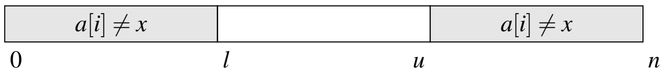
Abstract. Expressing linear integer constraints and assertions over integer ranges—as becomes necessary when reasoning about arrays—in a legible and succinct form poses a challenge for deductive program verification. Even simple assertions, such as integer predicates quantified over finite ranges, become quite verbose when given in basic first-order logic syntax. In this paper, we propose a domain-specific language (DSL) for assertions over integer ranges based on Reynolds's *interval* and *partition diagrams*, two diagrammatic notations designed to integrate well into linear textual content such as specifications, program annotations, and proofs. We extend intervalf diagrams to the more general concept of *mapping diagrams*, representing partial functions from disjoint integer intervals. A subset of mapping diagrams, *colorings*, provide a compact notation for selecting integer intervals that we intend to constrain, and an intuitive new construct, the *legend*, allows connecting colorings to first-order integer predicates. Reynolds's diagrams have not been supported widely by verification tools. We implement the syntax and semantics of partition and mapping diagrams as a DSL and theory extension to the Why3 program verifier. We illustrate the approach with examples of verified programs specified with colorings and legends. This work aims to extend the verification toolbox with a lightweight, intuitive DSL for array and integer range specifications.

1 Introduction

Deductive program verification is the activity of establishing correctness by mathematically proving verification conditions (VCs) extracted from a program and its specification. If all VCs are proved, the program is guaranteed to terminate in a state satisfying its postcondition for all inputs satisfying the precondition. While much of the mechanics of program verification is automated by VC generators and automatic theorem provers, the construction of correct programs by this method remains a largely interactive task. In the case of total correctness verification of sequential programs, VC generation relies on supplying a pre- and postcondition specification of each subroutine (procedure, method), and verification of a subroutine in turn requires intermediate assertions and loop invariants to be inserted into the routine. Producing these assertions requires both familiarity with a formal state description language and the ability to express the assertions in it succinctly. Such languages are usually based on first- or

E. Komendantskaya and Y. A. Liu (Eds.): PADL 2020, LNCS 12007, pp. 196–212, 2020.
https://doi.org/10.1007/978-3-030-39197-3_13

higher-order logic, and like programming languages general-purpose. While concise at expressing basic mathematical relations, constraints over arrays and integer ranges tend to require verbose expressions, obfuscating the original notion. Indeed, for array constraints, pictures often provide a more intuitive grip on the problem. For instance, given the textbook verification exercise of specifying the loop invariant of a binary search routine that determines the presence of the value x in a sorted array a (indexed from 0 to $n-1$), we may start by jotting down a box diagram similar to the following:

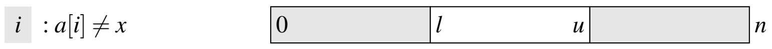


This diagram captures the pertinent assertions over the mutable state of binary search: that the loop variables l and u partition the array into three disjoint subarrays, and that the value x is not present in the leftmost or rightmost subarray. Once these relationships have been understood, we may then refine the diagram into a logic formula. A possible rendition of the above in first-order predicate logic is:

$$0 \leq l \leq u+1 \leq n \;\wedge\; \forall i(0 \leq i < l \vee u < i < n \Rightarrow a[i] \neq x)$$

It is easy to see that the formula lacks the legibility of the diagram, but is it actually more *formal*, as to make us accept this tradeoff? If we stipulate that the juxtaposition of the indexes (0, l, u and n) and the vertical lines denotes order constraints, and that the propositions written inside the shaded ranges are universally quantified over the corresponding subarrays, the diagram becomes semantically equivalent to the predicate logic formula. Hence, if the diagram incurs no loss of information, but appears more closely connected to our understanding of the domain, reasoning with the diagram directly could benefit both precision and legibility. As Dijkstra notes, "the purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise" [8]. However, unlike diagrams, predicate logic carries with it a collection of formula manipulation rules. Only given similar rules for diagrams like the above, may we consider them a worthy alternative to predicate logic for writing specifications and proofs. This is precisely the motivation behind Reynolds's *interval* and *partition diagrams*, introduced 40 years ago [16] together with a set of abstractions for reasoning about arrays. Reynolds argues "Of course, an equivalent assertion can be given in the predicate calculus, but this sacrifices the intuitive content of the diagram [...] A better approach is to formalize and give rigorous meaning to the diagram itself."

Approaching diagrams as formal specifications in their own right rather than as stepping stones, we do observe some rough edges in the box diagram: multiple occurrences of the expression $a[i] \neq x$ and the *ad hoc* assumption that i is quantified over the indexes of the shaded subarrays, while a and x, on the other hand, are free. To avoid the redundancy and clarify variable binding in the shaded subarrays, we redraw the diagram as follows:



The revised diagram consists of two components: a *legend* asserting $a[i] \neq x$ over a single shaded element at index i, and a box diagram specifying the constraints on l, u

and n as well as the extent of the shading. Following Reynolds, we also place the partition bounds inside the boxes rather than below them, as this convention both conserves space and increases legibility of a bound's position relative to its adjacent vertical line. Our intention with this example so far has only been to demonstrate that box diagrams are sufficiently precise for formal specification. As we will see through examples in the sequel, legends extend naturally to properties involving multiple indexes, such as sortedness (which we omitted for brevity in the above example). We give a detailed syntax and semantics of these diagrams, and present tool support for verifying programs specified with such diagrams.

Reynolds's Diagrams as a DSL. Domain-specific languages (DSLs), mini-languages tailored for a particular application, are commonly used in software engineering to raise the level of abstraction. The degree of sophistication range from substitution macros to rich high-level languages. A DSL should be easy to both use and opt out of if not deemed beneficial. Following the DSL approach, we decided to add diagram support to an existing language satisfying the following desiderata:

– Established in the verification community and supported by state-of-the-art, open-source tooling for VC generation and automatic theorem proving.
– Able to lexically combine box diagrams with the language's own syntax.
– Able to represent diagrams as data types, avoiding error-prone lexical translation stages and enabling use of diagrams in proofs.
– Tooling supports automatic reduction of VCs containing diagrams into their logical equivalences, e.g., by rewrite rules.

Consequently, we chose Why3 [11], an open-source platform[1] for deductive program verification consisting of a specification language, an ML-like programming language, and a common interface to multiple automatic theorem provers—including SMT solvers, the workhorses of modern program verification.

Contribution. We generalize Reynolds's interval diagrams to *mapping diagrams*, formally partial functions from a set of disjoint integer intervals to any type. *Colorings* constitute a subset of mapping diagrams, labeling intervals from a finite set ("palette") of colors. We introduce the *legend* construct for attaching interpretation to colorings. Intuitively, colorings specify labeled selections, e.g., "all integers between 0 and l are red" and "all indexes in the array are green", while legends express quantified predicates like "x is not among the red elements", "all red elements are greater than all green elements" and "all green elements are sorted". We show that colorings and legends are automatically reducible to universally quantified predicates. We have implemented an extension to the Why3 theorem prover to support diagrams similar to those shown in the introduction. The extension consists of a DSL allowing partition and mapping diagrams to be used in Why3 theories and programs, and a Why3 theory encoding partition and mapping diagrams as a data type together with the functions and predicates defining their semantics. The diagram syntax is character-based and does not require sophisticated editor support. All properties have been mechanically proved in Why3 using its underlying theorem provers. We demonstrate the DSL by verified code examples.

[1] Binary, source, and documentation available at https://why3.lri.fr.

Notational Conventions. We give the semantics of diagrams in first-order predicate logic with partial expressions. While the Why3 logic is total and requires a parametric data type (`option`) for expressing partiality, we describe the general semantics using partial expressions for brevity. We denote by the operator def definedness of a partial expression, and by $=_\exists$ the existential equality relation between two partial expressions e_1 and e_2 satisfying

$$e_1 =_\exists e_2 \;\hat{=}\; \mathsf{def}\, e_1 \wedge \mathsf{def}\, e_2 \wedge e_1 = e_2$$

In syntax definitions we adopt the convention that the meta-variables A and B stand for integer expressions, Q stands for Boolean expressions (predicates), E and F stand for expressions of other types, and X stands for identifiers. Subscript variables (e.g., A_1, A_2) are considered separate meta-variables of the same class. For sequences of identifiers, we write $\bar{X}$. We write $E^?$ to indicate that an expression may be omitted in a syntactic context, and we semantically handle absence by partiality. We indicate by $E[X]$ that X is a free variable in E. When $E[X]$ occurs as a subexpression we assume that adjacent subexpressions do not contain free occurrences of X; for instance, in the syntax definition $Q_1[X] \wedge Q_2$, the Boolean expression Q_1 may contain free occurrences of X whereas Q_2 may not. We write $\lambda X(E[X])$ for the anonymous function of variable X to value E. Other logic and arithmetic operators are the standard ones.

Overview of Paper. The rest of the paper is structured as follows. Section 2 describes interval and partition diagrams. Section 3 generalizes interval diagrams to mapping diagrams and colorings. Section 4 introduces the legend notation for assertions over colored intervals. Section 5 describes a tool extension allowing diagrams to be used in Why specifications. We illustrate use of this tool by example in Sect. 6. We review related work in Sect. 7 and conclude the paper with a discussion of lessons learned so far and possible future research directions in Sect. 8.

2 Interval and Partition Diagrams

Reynolds [16] introduces two interpretations for the pictogram $A_1\boxed{\qquad A_2}$: as an *interval diagram*, standing for the (possibly empty) integer interval $\{x \mid A_1 < x \leq A_2\}$, and as a *partition diagram*, standing for the predicate $A_1 \leq A_2$. This dual interpretation reflects the close semantical relationship between intervals and partitions. As diagrams are formulas, the intended meaning can in practice always be determined from the context: in a set-theoretic context it is an interval diagram, whereas in a logical context it is a partition diagram. Note that when $A_1 = A_2$ the partition diagram is universally true and the interval diagram represents the empty interval.

The form $A_1\boxed{\qquad A_2}$ is called the *normal form* of an interval or partition diagram, where both bounds are written to the left of the corresponding adjacent vertical lines, called *dividing lines*. Alternatively, either or both bounds of a diagram may be written to the right of the dividing line to offset the bound by 1. This means that the bound "$A-1|$" can be equivalently written as "$|A$". Below we list the alternative forms together with the corresponding normal forms and meanings as interval and partition predicate:

diagram	*equiv. normal form*	*integer interval*	*partition predicate*
$A_1\boxed{\qquad}A_2$	$A_1\boxed{\quad A_2-1}$	$\{x \mid A_1 < x < A_2\}$	$A_1 < A_2$
$\boxed{A_1 \qquad}A_2$	$A_1-1\boxed{\quad A_2-1}$	$\{x \mid A_1 \le x < A_2\}$	$A_1-1 \le A_2-1$
$\boxed{A_1 \quad A_2}$	$A_1-1\boxed{\quad A_2}$	$\{x \mid A_1 \le x \le A_2\}$	$A_1-1 \le A_2$

Note that when interpreted as partition diagrams, $A_1\boxed{\quad A_2}$ and $\boxed{A_1 \quad}A_2$ are equivalent, whereas when interpreted as interval diagrams, they represent different intervals. As a shorthand, we may write $\boxed{A}$ to denote the *singleton* interval containing only A:

$\boxed{A}$	$A-1\boxed{\quad A}$	$\{x \mid x = A\}$	$A-1 \le A$

When considered a partition diagram, $\boxed{A}$ is a tautology. However, the singleton form is still useful as a component of *general partition diagrams*. These consist of multiple chained partition diagrams that share dividing lines so that the right bound of the predecessor becomes the left bound of the successor. The following definition formalizes this notion.

Definition 1. *A* general partition diagram *is a sequence of n (where $n \ge 1$) component partition diagrams, with $n+1$ integer bounds $A_0,\dots,A_n$, asserting that these partition the total interval $A_0\boxed{\quad A_n}$ into n disjoint and connected component intervals:*

$$A_0\boxed{\quad A_1}\boxed{\quad A_2}\;\cdots\; A_{n-1}\boxed{\quad A_n} \quad \hat{=} \quad \bigwedge_{j=0}^{n-1} (A_j \le A_{j+1})$$

(Here the fragment $A_2\;\cdots\;A_{n-1}$ is meta-syntax standing for any number of intermediate component intervals; it is not part of the actual diagram syntax).

While each component diagram in Definition 1 is given on normal form (where each component interval bound is written to the left of the dividing line), as with the basic partition diagrams, a bound may be written on the opposite side of the dividing line to offset it by 1. We illustrate this with two examples.

Example 2.1. The partition diagram corresponding to the partial binary search invariant discussed in Sect. 1, $\boxed{0 \quad}\boxed{l \quad u}\boxed{\quad}n$, has the equivalent normal form $-1\boxed{\quad l-1}\boxed{\quad u}\boxed{\quad n-1}$ and stands for the predicate $0 \le l \le u+1 \le n$ (equivalently $-1 \le l-1 \le u \le n-1$).

Example 2.2. The partition diagram $\boxed{0 \quad}\boxed{k}\boxed{\quad n}$ has the equivalent normal form $-1\boxed{\quad k-1}\boxed{\quad k}\boxed{\quad n}$ and stands for the predicate $0 \le k \le n$ (equivalently $-1 \le k-1 \le k \le n$).

We note that Definition 1 is stricter than Reynolds's original definition, which considers the diagram true also when $A_0 \ge A_1 \ge \cdots \ge A_n$. I.e, in the original notation an empty partition may be specified with a left bound exceeding the right bound (i.e., $A_i > A_{i+1}$). Reynolds calls such diagrams *irregular* representations of the empty interval. Our definition allows only for what Reynolds refers to as *regular* representations of empty intervals (i.e., $A_i = A_{i+1}$), and a partition diagram is always false if any $A_i > A_{i+1}$. The stricter interpretation has the advantages that the basic partition diagram with two bounds $A_1\boxed{\quad A_2}$ constitutes a meaningful assertion by itself (rather than a tautology), and that the cardinality of an interval diagram is $A_2 - A_1$ when its corresponding partition diagram is true. Unlike for partition diagrams, Reynolds does not define a chained form for interval diagrams.

3 Mapping Diagrams and Colorings

Next we introduce *mapping diagrams*, a generalization of interval diagrams to partial functions from the integers. A mapping diagram consists of a sequence of *mapping components*. A mapping component $X \longrightarrow A_1\boxed{{}^{\#}E[X]\quad A_2}$, where E is a total expression over the integer parameter X to some type T, stands for the function $\lambda X(E)$ from the domain $A_1\boxed{\quad A_2}$ to the range T.

Definition 2. *A* general mapping diagram *is a sequence of mapping components that stands for the union of the corresponding functions:*

$$X \longrightarrow A_0\boxed{{}^{\#}E_0[X]^{?}\quad A_1}\ \cdots\ A_{n-1}\boxed{{}^{\#}E_{n-1}[X]^{?}\quad A_n}$$
$$\hat{=}$$
$$\bigcup_{i=0}^{n-1}\left\{(x, \lambda X(E_i)(x)) \mid x \in A_i\boxed{\quad A_{i+1}} \wedge \mathsf{def}\ E_i\right\}$$

We note that when the corresponding partition diagram $A_0\boxed{\quad A_1}\cdots A_{n-1}\boxed{\quad A_n}$ is true, the union of tuples is a partial function (as the domains of the component functions are disjoint). This is a side condition of the definition that we always verify when introducing a mapping diagram. In the diagram, an expression E_i may be omitted to indicate that the mapping is undefined on the interval $A_i\boxed{\quad A_{i+1}}$.

Property 3.1. A mapping diagram with bounds $A_0, \ldots, A_n$ and expressions $E_0, \ldots E_{n-1}$ is well-defined in each point of each interval i where E_i is present, and undefined in each point on each interval j where E_j is absent as well as in each point outside of the total interval $A_0\boxed{\quad A_n}$.

Example 3.1. The mapping diagram $k \longrightarrow a\boxed{{}^{\#}-k\quad b}\boxed{\quad c}\boxed{{}^{\#}k\quad d}$ stands for the following partial piecewise defined function: $\lambda k\begin{cases} -k & \text{if } a < k \le b \\ k & \text{if } c < k \le d \end{cases}$. The function is undefined on the interval $b\boxed{\quad c}$.

Definition 3. *A* coloring *is a mapping diagram where each component interval is either unmapped, or mapped to a member of a set of labels (colors) C:*

$$A_0\boxed{{}^{\#}E_0^{?}\quad A_1}\ \cdots\ A_{n-1}\boxed{{}^{\#}E_{n-1}^{?}\quad A_n}$$

In the above, each $E_i \in C$ if $\mathsf{def}\ E_i$.

The term *coloring* reflects the use of colors for marking intervals of interest in box diagrams. In particular, we will use colorings to assert a given predicate (specified with the *legend* construct described in the next section) over one or more intervals.

Property 3.2. A coloring col with component bounds $A_0,\ldots,A_n$ maps each point in an interval with a defined color to that value. That is, for each interval j, $0 \le j < n$:

$$\forall k \left(A_j < k \le A_{j+1} \wedge \mathsf{def}\ E_j \Rightarrow col(k) = E_j\right)$$

Example 3.2. The coloring $\boxed{0\ \ {}^{\#}\mathbf{R}}\boxed{l\qquad u}\boxed{{}^{\#}\mathbf{R}}\,n$ stands for the following piecewise defined partial function: $\lambda k \begin{cases} \mathbf{R} & \text{if } 0 \le k < l \\ \mathbf{R} & \text{if } u < k < n \end{cases}$.

4 Legends

A *legend* defines the interpretation of a coloring by a parametric, universally quantified assertion over all intervals colored in accordance with the legend.

Definition 4. *A* legend *is a binding expression over a sequence of integer variables $\bar X$ associating a coloring with bounds $A_0[\bar X],\ldots,A_n[\bar X]$ and colors $E_0,\ldots,E_{n-1}$ of type C to a predicate $Q[\bar X]$:*

$$\bar X \,:\, A_0[\bar X]\boxed{{}^{\#}E_0^{?}\quad A_1[\bar X]}\ \cdots\ A_{n-1}[\bar X]\boxed{{}^{\#}E_{n-1}^{?}\quad A_n[\bar X]} \longrightarrow Q[\bar X]$$

It stands for the following parametric predicate where the parameter $r \in \mathbb{Z} \to C$:

$$\forall \bar X \left(\bigwedge_{j=0}^{n-1} \left(A_j\boxed{\qquad A_{j+1}} \wedge (\mathsf{def}\ E_j \Rightarrow \forall k(k \in A_j\boxed{\qquad A_{j+1}} \Rightarrow r(k) =_{\exists} E_j))\right) \Rightarrow Q \right)$$

Informally put, the legend states that Q is true for a partitioning if the parameter function r returns the prescribed color value in every point of each colored component (on uncolored component intervals, the value of r is ignored).

Example 4.1. The legend "$i : \boxed{i\,{}^{\#}\mathbf{R}} \longrightarrow a[i] \ne x$" stands for the following parametric predicate over r:

$$\begin{aligned} & \forall i\,(i-1 \le i \wedge (\mathsf{def}\ \mathbf{R} \Rightarrow \forall k(i-1 < k \le i \Rightarrow r(k) =_{\exists} \mathbf{R})) \Rightarrow a[i] \ne x) \\ \equiv\ & \{\ \text{tautology elimination, singleton quantification domain}\ \} \\ & \forall i\,(r(i) =_{\exists} \mathbf{R} \Rightarrow a[i] \ne x) \end{aligned}$$

Example 4.2. The legend "$i\ j : \boxed{i\,{}^{\#}\mathbf{B}}\boxed{\qquad}\boxed{j\,{}^{\#}\mathbf{B}} \longrightarrow a[i] \le a[j]$" is equivalent to the following parametric predicate over r (the predicate has been simplified):

$$\forall\, i\ j\ (i < j \wedge r(i) =_{\exists} \mathbf{B} \wedge r(j) =_{\exists} \mathbf{B} \Rightarrow a[i] \le a[j])$$

Informally, Example 4.1 states that x is not among the elements of the array a colored **R**, while Example 4.2 states that the elements of a at ordered index pairs i, j colored **B** are sorted in nondecreasing order (regardless of coloring of interjacent indexes). To use a legend in expressing a state assertion, we apply it to a coloring function over the state space of the program we are specifying.

Example 4.3. Applying the legend given in Example 4.1 to the coloring $\boxed{0 \quad ^{\#}\mathbf{R}}\,l$ reduces to an assertion that the subarray $a[0],\ldots,a[l-1]$ does not contain the value x:

$$
\begin{aligned}
&(i : \boxed{i\, ^{\#}\mathbf{R}} \to a[i] \neq x)(\boxed{0 \quad ^{\#}\mathbf{R}}\,l\,)\\
\equiv\ &\{\ \text{Definition 4, } \beta\text{-reduction}\ \}\\
&\forall\, i\ ((\boxed{0 \quad ^{\#}\mathbf{R}}\,l\,)(i) =_{\exists} \mathbf{R} \Rightarrow a[i] \neq x)\\
\equiv\ &\{\ \text{definition of } =_{\exists}\ \}\\
&\forall\, i\ (0 \leq i < l \Rightarrow a[i] \neq x)
\end{aligned}
$$

An important design consideration has been that using legends and colorings when writing assertions should not result in formulas that make automatic verification more difficult compared to equivalent assertions written in traditional quantifier notation. As the inner quantification in the legend definition and the color type C are syntactic artifacts of the DSL, they should preferably be eliminated from the correctness formula before applying SMT solvers or other ATPs. To achieve this, our tool applies to each formula an elimination rule that rewrites terms of the form $lgd(col)$, where lgd is a legend and col is a coloring. The following proposition formalizes the elimination rule.

Proposition 1. *Given the legend*

$$lgd = \bar{X} : A_0[\bar{X}]\boxed{^{\#}E_0^{?} \quad A_1[\bar{X}]}\ \cdots\ A_{n-1}[\bar{X}]\boxed{^{\#}E_{n-1}^{?} \quad A_n[\bar{X}]} \to Q[\bar{X}]$$

and the coloring

$$col = B_0\boxed{^{\#}F_0^{?} \quad B_0}\ \cdots\ B_{m-1}\boxed{^{\#}F_{m-1}^{?} \quad B_m}$$

the following equivalence holds for the application $lgd(col)$:

$$lgd(col) \equiv \forall \bar{X}\left(\bigwedge_{j=0}^{n-1}\left(A_j \leq A_{j+1} \wedge (\mathsf{def}\ E_j \Rightarrow \mathsf{contains}\ (A_j, E_j, A_{j+1}, col))\right) \Rightarrow Q\right)$$

where $\mathsf{contains}$ *is defined recursively on the structure of mapping diagrams:*

$$
\begin{aligned}
&\mathsf{contains}\ (a, e, b, A\boxed{^{\#}E\ B}\boxed{}\,) \mathrel{\hat{=}} && \textit{(rec. case)}\\
&\quad b \leq a\\
&\quad \vee\ (A \leq a \leq B \wedge (A = B \vee e =_{\exists} E) \wedge \mathsf{contains}\ (B, e, b, B\boxed{}\,))\\
&\quad \vee\ \mathsf{contains}\ (a, e, b, B\boxed{}\,)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{contains}\ (a, e, b, A\boxed{^{\#}E \quad B}\,) \mathrel{\hat{=}} && \textit{(base case)}\\
&\quad b \leq a\\
&\quad \vee\ (A \leq a \leq B \wedge (A = B \vee e =_{\exists} E))
\end{aligned}
$$

Proof. $\Leftarrow$ by structural induction on col and Property 3.2, $\Rightarrow$ by transitivity of contains and induction over an integer interval.

Note that the definition of $\mathsf{contains}$ involves only Boolean connectives, integer comparison, and color terms of the form $e =_{\exists} E$. When e and E are literal color values (or absent), the equality $e =_{\exists} E$ can be immediately evaluated, reducing the inner quantification of the legend to a propositional formula where the atoms are linear integer constraints.

5 Diagram Extension to the Why3 Verification Platform

We have developed a prototype extension Why3 supporting mechanically proving meta-properties like Proposition 1 as well as specifying programs with partition and mapping diagrams. We first briefly present relevant features of the Why3 platform and then describe our implementation. The implementation is available in source form at https://gitlab.com/diagrammatic/whyp.

The Why3 Platform. The Why3 specification language is a first-order logic extended with polymorphic types and algebraic data types. Theorems can be proved using over a dozen supported automatic and interactive theorem provers. Users interact with Why3 in batch mode through a command-line interface, or interactively through a graphical IDE. In addition to the purely logical specification language, programs with loops, mutable variables, and exceptions can be written in the WhyML language. Why3 generates VCs from WhyML programs based on weakest preconditions. A good example-driven tour of Why3 is given by Bobot et al. [6]. Why3 provides a set of transformations on verification tasks, including definition inlining, application of rewrite rules and logical inference rules. The Why3 drivers for external theorem provers employ these to transform the sequent into a form amenable to automation, for instance by eliminating language features that are not supported by the target prover (e.g. algebraic data types). Other transformations are intended to be used interactively to reduce a goal into one or more subgoals, e.g., proof-by-cases, instantiation, and induction. Why3 is designed to both be a backend for other tools as well as an extensible verification platform. It comes with a standard library of theories which can be reused through an importing mechanism. The platform itself can be extended with different kinds of plug-ins adding new input formats, prover drivers and other extensions. The core of Why3 is implemented in OCaml and is offered as a software library for programmatic access to all platform functionality, including parsing, typing, transformations, and invocation of external theorem provers.

Extension Architecture. The extension to Why3 consists of two components: a set of Why3 theories formalizing partition diagrams and mappings, and a syntactic preprocessor (written in OCaml) that translates the concrete diagram syntax into Why3 terms. Figure 1 shows the data flow when the user asks the tool to check a theory containing diagrams (indicated by the theory existing in a file with the suffix `.whyp`). The preprocessor parses the input theory and translates all partition diagrams, mapping diagrams and legends into normal form and then into instances of a data type defined in the theory extension. The resulting AST is dispatched to Why3 for typing, inclusion of standard library theories, task generation, and external theorem prover execution. From here onwards the data flow is identical to that of checking a normal Why3 theory. Next, we describe the concrete diagram syntax and the embedding of the diagrams in the Why3 logic.

DSL Syntax and Semantics. The DSL follows the ASCII-based lexical rules of Why3 [5]. A partition diagram must be enclosed in square brackets '[' and ']', and vertical dividing lines are written as '`|`'. The leftmost or rightmost vertical line may be omitted

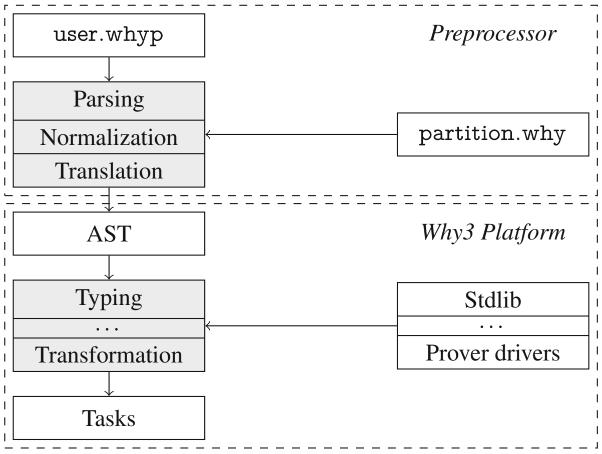
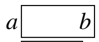
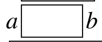
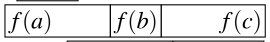
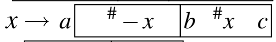
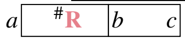


Fig. 1. Processing pipeline of Why3 extension

when it would be adjacent to a square bracket. The bounds themselves follow the syntax of Why3 terms. Mapping diagrams must be enclosed in '[X→[' and ']]' (the binder X may be omitted for colorings). The ellipsis '…' separates bounds inside a component, and in mapping diagrams may be followed by '#' and an expression. The following are examples of accepted ASCII partition and mapping diagrams and their diagrammatic equivalents:

ASCII syntax	*diagrammatic syntax*
`[a| … b]`	
`[a| … |b]`	
`[f a … | f b | … f c]`	
`[x → [a| …#(-x) |b …#(x) c]]`	
`[[a| … #R |b … c]]`	

Partition and mapping diagrams may occur in a Why3 theory anywhere a term is expected. After parsing, both types of diagrams are translated into instances of the polymorphic data type diag:

```
type diag α =  P int (option α) (diag α) | L int (option α) int
```

The data type represents partition diagrams in normal form; the preprocessing hence includes a normalization stage, converting each '|*e*'-fragment into '*e*-1|' and each '|*e*|'-fragment into '*e*-1|…*e*|', before finally converting the result into an instance of the above data type. The option data type from Why3s standard library is used to handle partiality. It has two constructors, None and Some α. For partition diagrams, the second parameter is always None. For mapping diagrams, it is Some α for each interval associated with an expression of type α, otherwise None. The semantics of partition diagrams is given by the predicate partitioning:

```
predicate partitioning (d:diag α) = match d with
  | P a _ ((P b _ _ | L b _ _) as r) → a≤b ∧ partitioning r
  | L a _ b → a≤b
  end
meta "rewrite_def" predicate partitioning
```

and for mapping diagrams by the function mapping:

```
function mapping (d:diag α) (i:int) : option α = match d with
  | P a e ((P b _ _ | L b _ _) as r) → if a<i≤b then e else mapping r i
  | L a e b → if a<i≤b then e else None
  end
meta "rewrite_def" function mapping
```

The **meta** declarations instruct Why3 to use the above definitions as rewrite rules when transforming a proof task in preparation for sending it to an external theorem prover. The rewrites are applied recursively and exhaustively, viz. partitioning is rewritten into a conjunction sequence and mapping into a nested **if-else**-expression. This is normally desirable when using diagrams for specification; only when proving meta-theorems about partition and mapping diagrams may we want to suppress automatic rewriting.

A legend is declared with the **legend** keyword followed by an identifier, a sequence of parameters and a semicolon-separated list of coloring-to-predicate mappings. The preprocessor translates the legend into a conjunction of universally quantified statements according to Definition 4. For example, the legend:

```
type col = R | G
legend lgnd(a: array int)(x:int) of col =
  i : [[i#R]] → a[i]≠x;
  i,j : [[i| ... #G j]] → a[i]≤a[j]
```

is translated by the preprocessor into the following Why3 predicate:

```
predicate lgnd(a:array int)(x:int)(~r:int→option col) =
  (forall i:int. i-1≤i ∧ (forall ~k:int. i-1<~k≤i → ~r ~k = Some R) → a[i]≠x) ∧
  (forall i,j:int. i≤j ∧ (forall ~k:int . i<~k≤j → ~r ~k = Some G) → a[i]≤a[j])
```

For automatic elimination of the inner quantification and color-typed terms in a legend applied to a coloring, as described in Sect. 4, the partitioning theory includes the following lemma declared as a rewrite rule:

```
lemma mapping_to_contains [@rewrite]:
  forall a,b:int, c:α, d: diag α.
    (forall k:int. a<k≤b → mapping d = Some c) ↔ contains a b c d
```

Here contains is defined as in Proposition 1. The rewrite is automatically applied (from left to right) by Why3 when executing the compute_specified and compute_in_goal transformations on a goal. The default strategy of the extended Why3 verifier applies these transformations prior to invoking the user's back-end theorem prover of choice.

Listing 1. Dutch National Flag

```
type col = B | W | R

legend flag(a: array col) =
  i: [[i#B]] → a[i] = B;
  i: [[i#W]] → a[i] = W;
  i: [[i#R]] → a[i] = R;

let dutch_national_flag (a: array col) : unit
  ensures { exists b r: int . flag a [[0 ...#B |b ...#W |r ...#R |length a]] }
  ensures { permut_all (old a) a }
=
  let b = ref 0 in
  let i = ref 0 in
  let r = ref (length a) in
  while !i < !r do
    invariant { [0 ... |!b ... |!i ... |!r ... |length a] }
    invariant { flag a [[0 ...#B |!b ...#W |!i ... |!r ...#R |length a]] }
    invariant { permut_all (old a) a }
    variant { !r - !i }
    match a[!i] with
    | B → swap a !b !i;  b := !b + 1; i := !i + 1
    | W → i := !i + 1
    | R → r := !r - 1; swap a !r !i
    end
  done
```

6 Verified Code Examples

In this section we present three formally specified and verified procedures where the pre- and postconditions and loop invariants are expressed as diagrams. The procedures are specified in the WhyML language with the diagram extensions described in Sect. 5, and all VCs were proved automatically by a combination of Z3 [7], CVC4 [3] and Alt-Ergo [4] after preprocessing by our tool.

A classical example of using the coloring analogy in verification is the *Dutch National Flag* problem introduced by Dijkstra [9]. It is a simplified sorting problem: an array containing, in random order, any number of each of the three values blue (B), white (W), and red (R) should be rearranged so that the blue elements precede all the white elements, which in turn precede all the red elements (i.e., the final order is B, W, R). Listing 1 shows an adaptation of an existing Why3 solution[2], in which we have replaced the textual postcondition (**ensures** clauses) and loop invariant (**invariant** clauses) with diagrammatic equivalents. The procedure executes in time linear to the size of the array and mutates the array by pairwise compare and swap. The loop invariant consists of three components: a partition diagram constraining the values

[2] Part of a gallery of verified programs available at http://toccata.lri.fr/gallery/why3.en.html.

Listing 2. Binary search

```
type sorted_col = SO

legend sorted (a: array int) of sorted_col =
   i,j: [[i#SO| ... |j#SO]] → a[i] ≤ a[j] ;

type found_col = NE | EQ

legend found (a: array int) (x:int) of found_col =
   i: [[i#NE]] → a[i] ≠ x ;
   i: [[i#EQ]] → a[i] = x

let binary_search (a: array int) (x: int) : int
   requires { sorted a [[0 ...#SO |length a]] }
   ensures { [0 ... |result| ... |length a] }
   ensures { found a v [[result#EQ]] }
   raises { Not_Found → found a x [[0 ...#NE |length a]] }
=
   let l = ref 0 in
   let u = ref (length a - 1) in
   while !l ≤ !u do
      invariant { sorted a [[0 ...#SO |length a]] }
      invariant { [0 ... |!l] ∧ [!u ... |(length a)-1] }
      invariant { found a x [[0 ...#NE |!l ... !u| ...#NE |length a]] }
      variant {!u - !l}
      let m = !l + div (!u - !l) 2 in
      if a[m] < x then l := m + 1
      else if a[m] > x then u := m - 1
      else return m
   done;
   raise Not_Found
```

of the loop variables `b`, `i` and `r`; a coloring mapping the intervals $\boxed{\mathtt{0}\qquad}\mathtt{b}$, $\boxed{\mathtt{b}\qquad}\mathtt{i}$ and $\boxed{\mathtt{r}\qquad}\mathtt{length\ a}$ to B, W and R, respectively; and a (non-diagrammatic) assertion that the modified array is a permutation of the original. The `swap` operation and the `permut_all` predicate are imported from the Why3 array library together with the property that the former maintains the latter. The program is atypical in that the color values, represented by the datatype `col`, are not pure specification constructs but also occur in the computation itself. The legend `flag` is trivial due to the nature of the program; it simply asserts that `a` is elementwise equal to the coloring on the intervals on which the latter is defined.

Listing 2 shows a verified implementation of binary search with a diagrammatic postcondition and loop invariant similar to the invariant discussed in the introduction. The procedure `binary_search` determines the presence of the value `x` in the sorted input array `a`. It has two exits, one normal and one abnormal. If `x` is found in `a`, the procedure

Listing 3. Insertion sort

```
type col = SO | I

legend sorting (a: array int) of col =
   i,j: [[i#SO| ... |j#SO]] → a[i] ≤ a[j] ;
   i: [[i#I|i+1#SO]] → a[i] ≤ a[i+1]

let insertion_sort (a: array int)
   ensures { sorting a [0 ...#B |length a] }
   ensures { permut_all a (old a) }
=
   let m = ref 0 in
   while !m < length a do
      invariant { [0 ... |!m| ... length a] }
      invariant { sorting a [[0 ...#SO |!m]] }
      invariant { permut_all a (old a) }
      variant { length a - !m }
      let k = ref !m in
      while !k > 0 && a[!k-1] > a[!k] do
         invariant { [0 ... |!k| ... !m] }
         invariant { sorting a [[0 ...#SO |!k#I| ...#SO !m]] }
         invariant { permut_all a (old a) }
         variant { !k }
         swap a !k (!k - 1);
         k := !k - 1
      done;
      m := !m + 1
   done;
```

exits normally returning an index containing x (in Why3, normal return values are represented by the result variable in the postcondition specification). If x is not found in a, the procedure exits abnormally in the Not_Found exception carrying the associated postcondition that all elements of a are different from x. To express the specification and invariants diagrammatically, we introduce two legends for the specification of binary search: sorting, expressing sortedness of the SO-colored range; and found, expressing existence (EQ) or absence (NE) of the sought element x.

The final example, Listing 3 shows an implementation of a simple sorting algorithm, insertion sort. The procedure insertion_sort sorts the input array by maintaining two partitions, one sorted partition ranging from index 0 to m, followed by one unsorted partition ranging from m to the end of the array. Each iteration of the outer loop extends the sorted partition by one element, until the whole array is sorted. The outer loop invariant is expressed by the partition diagram |0 |!m| length a| and the coloring |0 #SO |!m. The first component of the legend sorting for SO-colored intervals is identical to the one introduced in Listing 2. In order to achieve sortedness of the first interval after incrementing m, the inner loop moves the element at index m backwards into its final position in the sorted partition by repeatedly swapping it with its predecessor using the loop counter `k`. During the execution of the inner loop, the outer loop invariant is temporarily invalidated: the interval $\boxed{\texttt{0}\quad\texttt{!m}}$ is almost sorted, but with the exception of a single potential inversion of elements at indexes `k-1` and `k`. Diagrammatically, this is expressed by the index `k` being colored by `I`, and the second component of the `sorting` legend specifying that any `I`-colored element followed by an `SO`-colored element constitutes a sorted pair.

7 Related Work

Partition and interval diagrams were originally proposed by Reynolds [16], who used them extensively in the specification and verification of several array-manipulating programs in his textbook *The Craft of Programming* [17]. Notably, Reynolds gives a formal syntax and a set of manipulation rules for the diagrams to facilitate their use as terms in calculational correctness proofs. Reynolds writes universally quantified invariants using the standard ∀-operator and gives the quantification domain as an interval diagram, rather than making the quantification implicit in the diagrammatic notation itself. Many textbooks, e.g. [12,18], use similar box diagrams (termed "array pictures" or "array diagrams") in the presentation of assertions over both array indexes and array elements, but most of them do not formalize their semantics fully. Astrachan [1] suggests diagram representations for arrays and linked lists, emphasizing the role of diagrams in comprehension. Generating visual representations from textual specifications has been addressed in the context of the Z language [14]. Similar diagrams have also been proposed for visualizing array VCs [13], as an aid to proof and debugging. Wickerson et al. [19] employ partition-like diagrams in the visual proof notation called ribbon proofs for separation logic. Pearce [15] explores through a number of examples how array-based programming is enhanced by languages which support specifications and invariants over arrays. Invariant-based programming [2] is a correct-by-construction formal method aimed at teaching in which programs and their proofs are constructed diagrammatically, often with the aid of partition diagram-like pictures during the initial stages of construction. The idea of colorings and legends also originates from the authors' previous joint work with R-J. Back [10]. We are not aware of existing work on integrating partition or interval diagrams into a general-purpose program verification platform.

8 Conclusions and Future Work

This paper approaches box diagrams from the viewpoint that they can serve as an expressive formal mini-language—a DSL—rather than being restricted to their traditional role of ephemeral pre-code sketches and post-code visualizations. We have introduced an extension to Reynolds's original partition and interval diagrams for piecewise definition of partial functions over integer intervals, and a legend construct for asserting a predicate over a sequence of labeled ("colored") intervals. We have extended Why3 to read diagrammatic syntax and formalized its semantics in a Why3 theory.

We believe that the value of a formal specification is largely dependent on its legibility, as the specification cannot be proved correct (only checked for internal consistency), but is instead subject to human assessment of fitness for purpose (validation). Also, although significant advances have been made in automatically synthesizing invariants from the code, such techniques cannot discover difficult invariants, meaning that the same requirement of legibility applies to these as well. Good notation makes writing readable assertions more tractable, and while notation by itself may not be the primary challenge of verification, it is nevertheless held in high regard among verification practitioners. In particular, we have observed a tendency among the experienced to write integer range and array predicates following idioms that serve similar organizational purposes as the diagrams proposed herein, such as chaining relational operators and maintaining increasing order of bounds from left to right, preferring the relations $<$ and $\leq$ over $>$ and $\geq$. Formalizing such idioms is where a DSL and tool support can be of value. However, diagrammatic languages requiring sophisticated tool support and considerable learning investments from users are hard to justify for niche domains. The authors believe that the DSL presented here achieves a sensible compromise between expressive power and ease of integration. During implementation of the tool support, we came to appreciate both the semantically rich Why3 and WhyML languages and the extensibility of the Why3 platform through its API. The one feature we missed was a way to extend the Why3 language in a modular fashion, e.g., by adding new term productions while leaving the rest of the grammar intact, rather than by modifying its parser.

Finally, we emphasize that our goal has not been to address all aspects of array reasoning. The authors have found the DSL useful when writing array invariants that involves partitioning, but do not make further claims regarding its general applicability. There are classes of array invariants for which the DSL is not a sensible choice: clearly, assertions not involving partitions may have little to gain, and assertions with multiple nested quantifications may find the legibility advantage being lost. A lightweight DSL has the advantage that we can restrict its use to specification tasks for which we deem it beneficial, and fall back to standard FOL notation in other cases. Hence we believe the DSL's primary role is to ease writing of certain classes of array and integer range properties, viz. those that involve partitioning and universal quantification. We surmise that the class of programs involving such properties is large enough that a DSL like the one presented here could justify its place in the modern verification toolbox.

Future Work. There is scope for much further work on partition, interval and mapping diagrams, both in improving tool support and in generalizing the notation itself. The tool is in the first prototype stage and has so far only been tested on a small collection of toy examples. We have identified enhancements and optimizations that will be required to address real-world requirements. For instance, the diagram syntax is currently not supported in the Why3 IDE during interactive proofs. Also, the `contains` rewrite should be optimized, as it currently expands a diagram into a formula that is exponential in the length of the diagram chain (this has not been an issue in practice with typical diagrams, but prevents larger diagrams from being processed). More experiments will be needed to gain more experience with the notation and identify potential pitfalls, and a comparative study with real users is necessary to experimentally assess the merits of the DSL over regular FOL notation. Finally, we are also looking into extending the diagram notation beyond the domain of integers, in particular to non-linear structures in order to reason about multi-dimensional arrays, trees and graphs.

References

1. Astrachan, O.L.: Pictures as invariants. In: Dale, N.B. (ed.) Proceedings of the 22nd SIGCSE Technical Symposium on Computer Science Education, pp. 112–118. ACM (1991). https://doi.org/10.1145/107004.107026
2. Back, R.J.: Invariant based programming: basic approach and teaching experiences. Form. Asp. Comput. **21**(3), 227–244 (2009). https://doi.org/10.1007/s00165-008-0070-y
3. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo Automated Theorem Prover (2008). http://alt-ergo.lri.fr/
5. Bobot, F., Filliâtre, J.C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 Platform (2019). Version 1.2.0. http://why3.lri.fr/manual.pdf
6. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verifythis with why3. Int. J. Softw. Tools Tech. Transf. **17**(6), 709–727 (2015). https://doi.org/10.1007/s10009-014-0314-5
7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Dijkstra, E.W.: The humble programmer. Commun. ACM **15**(10), 859–866 (1972). https://doi.org/10.1145/355604.361591
9. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Upper Saddle River (1976)
10. Eriksson, J., Parsa, M., Back, R.-J.: A precise pictorial language for array invariants. In: Furia, C.A., Winter, K. (eds.) IFM 2018. LNCS, vol. 11023, pp. 151–160. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98938-9_9
11. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
12. Gries, D.: The Science of Programming, 1st edn. Springer, New York (1987)
13. Jami, M., Ireland, A.: A verification condition visualizer. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 72–86. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_5
14. Moremedi, K., van der Poll, J.A.: Transforming formal specification constructs into diagrammatic notations. In: Cuzzocrea, A., Maabout, S. (eds.) MEDI 2013. LNCS, vol. 8216, pp. 212–224. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41366-7_18
15. Pearce, D.J.: Array programming in whiley. In: Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, pp. 17–24. ACM, New York (2017). https://doi.org/10.1145/3091966.3091972
16. Reynolds, J.C.: Reasoning about arrays. Commun. ACM **22**(5), 290–299 (1979). https://doi.org/10.1145/359104.359110
17. Reynolds, J.C.: The Craft of Programming. Prentice Hall PTR, Upper Saddle River (1981)
18. Tennent, R.D.: Specifying Software - A Hands-On Introduction. Cambridge University Press, Cambridge (2002)
19. Wickerson, J., Dodds, M., Parkinson, M.: Ribbon proofs for separation logic. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 189–208. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_12