A High-level Derivation of Global Search Algorithms (with Constraint Propagation)

Peter Pepper
Douglas R. Smith

February 23, 1996

Abstract

In this paper we describe the formal derivation of a transportation scheduling algorithm. The algorithm is based on the concepts of global search and constraint propagation and was originally derived using KIDS (Kestrel Interactive Development System). The emphasis in this paper is on clarity of the overall derivation and on expressing concepts at a level of abstraction that permits significant reuse of concepts, laws, inference patterns, etc.

1 Introduction

This paper describes the formal derivation of a transportation scheduling algorithm. The algorithm is based on the concepts of global search and constraint propagation and has been derived using KIDS (Kestrel Interactive Development System) [8, 10, 12].

There are several reasons for focusing on the derivation of scheduling algorithms. First, scheduling is an important and difficult problem. Tremendous benefits arise from having good scheduling algorithms, because scheduling is concerned with the efficient use of scarce resources in carrying out the complex activities of an organization. Unfortunately, many practical scheduling problems are NP-hard, so it is unlikely that there are solution methods that are both general and efficient. The intrinsic combinatorial difficulty of scheduling practically requires heuristic algorithms for solving large-scale problems – optimal schedules can only be obtained for problems involving tens or hundreds of activities. Yet the suboptimal schedules produced by most schedulers means that time, money, and resources are wasted.

Second, the formal derivation of efficient scheduling algorithms is challenging and has forced us to generalize our previously developed theory of global search and to develop new techniques for deriving constraint propagation code [12]. The transportation scheduling problem treated in this paper was an early version of several schedulers developed by Kestrel for various organizations (e.g. [3]). KIDS has been used to generate some schedulers that are dramatically faster than comparable currently deployed schedulers. Other schedulers have been developed that handle constraints that have not previously been handled.

The emphasis in this paper is on clarity of the overall derivation and on expressing concepts at a level of abstraction that permits significant reuse (e.g. of theories, concepts, laws, and inference patterns). This derivation differs from previous ones [10, 12] in several ways. First, global search is presented in terms of operations on sets, rather than on representations of sets (cf. [7]), thereby simplifying and clarifying the presentation. Second, the specification of the scheduling problem makes use of higher-order functions and predicates, thereby revealing common structure between the constraints in the postcondition. We found that a small collection of higher-order predicates accounts for almost all common constraints arising in our scheduling applications. Furthermore, these predicates have simple laws and regular patterns of inference with respect to calculating propagation code.

Example 1 ***Transportation scheduling:*** *Transportation scheduling is an extremely rich application domain with a plethora of variations and complications. It turns out, however, that for the purposes of our conceptual presentation the following simplified instance covers many principal aspects: We are given a set of cargo items, the "movement requirements", and we have to schedule trips (of airplanes, or ships, etc.) for their transportation between two given sites. Each cargo item has a time when it is available at the earliest, and a time when it is due at the latest. These two times determine the "start window" of the movement. The trips have a certain (round-trip) duration and a limited capacity. These requirements are informally summarized in Figure 1. (A more precise – and thus longer – formalization is given in Figure 16 in the Appendix.)*

$\{\ cargo \mid cargo \in load(trip), trip \in schedule\ \} = CargoOnStock$	goal
$start\text{-}time(trip) \in start\text{-}window(cargo)$ **for all** $cargo \in load(trip)$	constraints
$start\text{-}time(trip) \geq start\text{-}time(previous(trip)) + roundtrip\text{-}time$	
$capacity(trip) \geq sum\{\ size(cargo) \mid cargo \in load(trip)\ \}$	
$vessel(trip) \in Fleet$	

Figure 1: Informal description of a transportation problem

In this paper algebraic specifications are used to present datatypes, application domain theories, problem theories, and algorithm theories (cf. [9]). In Section 2.1 we present basic specifications for various datatypes, and in Section 2.2 a specification for the concepts and constraints of our transportation scheduling problem. In Section 2.3 we specify a simple theory of problems, and in Section 4 an enumeration theory from which a form of global search theory can be composed. In Section 3 we describe the notion of global search as a method for enumerating a set and show how to derive a scheduler by driving problem constraints into the enumerator of the output type. Driving constraints into an enumerator results in an enumerator of a smaller set whose elements satisfy the constraints by construction. One particularly difficult class of constraints results in the use of constraint propagation code in the enumerator (Section 3.5).

The derivation presented here takes less than 10 minutes to perform on KIDS. The user interactively applies an algorithm design tactic to the scheduling specification, and applies a small number of optimizing transformations to the developing algorithm. The resulting algorithm is expressed in the Refine language and is compiled into CommonLisp. More than 70 applications have been developed using KIDS, most of which were *not* scheduling problems. Almost all the work required in these examples lies in specifying the concepts and laws of the application domain. In an unfamiliar domain, this involves learning about the domain and interacting with domain experts. This domain modeling is a necessary and irreducible aspect of (formal approaches to) software engineering. The scheduling domain theory for the simple problem presented here evolved over a period of several months. Part of the difficulty on specifying an application domain lies in providing the laws necessary for effective inference in the domain. Basic axioms alone do not suffice and often it is only after experimentation that the domain modeler can develop the higher-level lemmas needed by the theorem-prover to generate useful results. Current work at Kestrel is focused on building libraries of scheduling-specific theories that can be reused in specifying new scheduling applications.

2 Algebraic Framework for Problems and Solutions

As mentioned in the introduction, there are two aspects to our presentation: On the one side there is the *conceptual modeling* of our approach to global search and in particular to scheduling, and on the other side there is the *technical representation* of the various concepts that are used in the approach. In this section we briefly comment on the latter. We use a unified algebraic approach for specifying our application problems as well as for representing the algorithm theories and strategies that are used for the derivation of solutions. Program development then is mainly based on suitable compositions of specification diagrams taken from a library. Since we are mainly interested in the presentation of concepts, we do not want to be overly hampered by technicalities. Therefore we use a relatively free-style notation, which is however, strongly influenced by the languages OPAL[1] and SLANG[2].

2.1 Specification of Data Types

In this paper we are mainly concerned with two data types, namely *pairs* and *maps*. Therefore we briefly give their definitions in Figures 2 and 3. The notation should be mostly self-explanatory.[3] The type declaration for `pair` in Figure 2 describes a so-called free type. It comprises, in this case, a sort `pair` together with the constructor function $\langle \_ , \_ \rangle$ (here in mixfix notation) and two selector functions π_1 and π_2.

SPECIFICATION `Pair`[α , β]
SORT α -- *parameter sort* SORT β -- *parameter sort*
TYPE `pair` $==$ $\langle \_ , \_ \rangle(\pi_1 : \alpha , \pi_2 : \beta)$
AXM $\pi_1\langle$a , b$\rangle$ $==$ a AXM $\pi_2\langle$a , b$\rangle$ $==$ b

Figure 2: Specification of pairs

Figure 3 presents an excerpt from the specification of the data structure `Map`. In this structure, maps are built up using the *composition operator* $\boxplus$, starting from from *singleton maps* `[a` $\mapsto$ `b]`. For simplicity we restrict ourselves here to a variant, where composition is only allowed for maps with disjoint domains. This allows us to make composition not only associative but also commutative.

Higher-order predicates on maps

For our approach to global-search algorithms we need a number of predicates on maps that are best expressed by way of higher-order functions. They are collected in the specification `Map-Predicates` in Figure 4.

- The predicate `pointwise(p)` tests whether `p(a , b)` holds for all elements `a` , `b` with `b` $=$ `M . a`.

[1]OPAL is a language for algebraic specification and functional programming; it has been developed and implemented at the Technical University Berlin for doing experiments with high-level programming styles [5].

[2]SLANG is an algebraic specification language that is developed at the Kestrel Institute as part of the SPECWARE environment [13].

[3]To ease reading, we use the convention that equations hold whenever all their subexpressions are well-defined.

```
SPECIFICATION  Map[α , β]
--------------------------------------------------------------
  SORT α      -- parameter sort
  SORT β      -- parameter sort
--------------------------------------------------------------
  IMPORT  Set[α]
          Set[β]
--------------------------------------------------------------
  SORT  map
  FUN  [ _↦_ ] : α × β → map         -- singleton map
  FUN  _ ⊞ _ : map × map → map        -- composition of maps
  FUN  _ . _: map × α → β            -- selection
  FUN  domain _ : map → set[α]
  FUN  range _ : map → set[β]
      ...
--------------------------------------------------------------
  AXM  [a ↦ b].a  ==  b
  AXM  domain[a ↦ b]  ==  {a}
  AXM  M1 ⊞ M2 REQUIRES domain(M1) ∩ domain(M2) = ∅
  AXM  domain(M1 ⊞ M2)  ==  domain(M1) ∪ domain(M2)
  AXM  (M ⊞ [a ↦ b]).a  ==  b
  AXM  (M ⊞ [a ↦ b]).a′  ==  M.a′   IF  a ≠ a′
                                              ...
  AXM  M1 ⊞ (M2 ⊞ M3)  ==  (M1 ⊞ M2) ⊞ M3
  AXM  M1 ⊞ M2  ==  M2 ⊞ M1
```

Figure 3: Specification of maps

- The predicate `pointwise-on-range(p)` tests whether `p(b)` holds for every element `b` in the range of the map; analogously for `pointwise-on-domain`.
- The predicate `pointwise-on-inverse-map(p)` tests whether the predicate `p` holds for every set of those domain elements that are mapped to the same target element.
- The predicate `pairwise-on-range(p)` is the most complex: It tests whether the predicate `p(b`$_1$ `, b`$_2$`)` holds for any two elements `b`$_1$, `b`$_2$ in the range of the map.

2.2 Specification of the Scheduling Problem

The above specification technique is not only suited for describing data structures in a highly abstract fashion, but it can equally well be used to specify programming tasks to be solved. Figure 5 contains a formal specification of the transportation scheduling problem that was informally described in Figure 1 in the Introduction. This formal specification is given in terms of traditional pre- and postconditions. The postcondition is based on five predicates that are imported from the specification `Scheduling-Basics` (see Figure 16).

- The predicate `Fit` says that the start time of a trip must conform to the start window of all its cargo items as determined by availability and due time.

```
SPECIFICATION Map-Predicates[$\alpha$ , $\beta$]

    SORT $\alpha$
    SORT $\beta$

    IMPORT Map[$\alpha$ , $\beta$]

    FUN pointwise : ($\alpha \times \beta \rightarrow$ bool) $\rightarrow$ map $\rightarrow$ bool
    FUN pointwise-on-domain : ($\alpha \rightarrow$ bool) $\rightarrow$ map $\rightarrow$ bool
    FUN pointwise-on-range : ($\beta \rightarrow$ bool) $\rightarrow$ map $\rightarrow$ bool
    FUN pointwise-on-inverse-map : ($\beta \times$ set[$\alpha$] $\rightarrow$ bool) $\rightarrow$ map $\rightarrow$ bool
    FUN pairwise-on-range : ($\beta \times \beta \rightarrow$ bool) $\rightarrow$ map $\rightarrow$ bool

    AXM pointwise(p)(M) == ($\forall$ a $\in$ domain(M) . p(a , M . a))
    AXM pointwise-on-domain(p)(M) == ($\forall$ a $\in$ domain(M) . p(a))
    AXM pointwise-on-range(p)(M) == ($\forall$ b $\in$ range(M) . p(b))
    AXM pointwise-on-inverse-map(p)(M) == ($\forall$ b $\in$ range(M) . p({a | M . a = b}))
    AXM pairwise-on-range(p)(M) == ($\forall$ $a_1$ , $a_2$ $\in$ domain(M) . p(M . $a_1$ , M . $a_2$))
```

Figure 4: Specification of higher-order predicates on maps

- The predicate `Sep` says that successive trips must be sufficiently separated to allow for the round-trip duration.
- The predicate `Cap` simply says that the load of every trip cannot exceed the vessel's capacity.
- The predicate `Fleet` says that only available vessels from the given fleet can be used.
- Finally, the predicate `Complete` requires that all cargo to be moved is actually moved (and nothing else).

Note that we use an infix notation for set comprehension: `{p}` specifies the (possibly infinite) set of all elements `x`, for which the predicate `p(x)` holds. (This also motivates the Currying of the predicate `Post`). The last axiom in Figure 5 specifies the `Scheduling` function in terms of its pre- and post-conditions.

In order not to interrupt the course of the paper too much at this point we defer the formal presentation of these five predicates to the places where they are actually used. For easier reference the full specification is listed in Figure 16 in the Appendix.

2.3 Development Libraries

From a technical point of view, the core of the method is a library of predefined theories about problem structure and problem-solving methods, together with means for applying them to concrete problems. These theories are essentially represented in the same manner as the concrete problem specifications.

An elementary theory of `Problem` structure is presented in Figure 6. It is mainly used for "gluing together" more interesting theories and specifications. A "problem" is viewed as the task of implementing a function that is specified by its pre- and postcondition. For the purpose of this paper we consider only set-valued functions, that is, functions that meet specifications of the kind "return the set of all elements, for which … holds".

```
SPECIFICATION Scheduling
─────────────────────────────────────────────────────────────
  IMPORT Scheduling-Basics ONLY schedule movement
                                Fit Sep Cap Fleet Complete
─────────────────────────────────────────────────────────────
  FUN Scheduling : set[movement] → set[schedule]
  FUN Pre :        set[movement] → bool
  FUN Post :       set[movement] → schedule → bool
─────────────────────────────────────────────────────────────
  AXM Pre(Cargo) == (Cargo ≠ ∅)
  AXM Post(Cargo)(schedule) ==   Fit(schedule)
                               ∧ Sep(schedule)
                               ∧ Cap(schedule)
                               ∧ Fleet(schedule)
                               ∧ Complete(Cargo)(schedule)
─────────────────────────────────────────────────────────────
  AXM Pre(Cargo) ⇒ Scheduling(Cargo) = {Post(Cargo)}
```

Figure 5: Specification of the scheduling problem

Again, the notation should be self-explanatory. On the level of abstraction chosen in this paper the keywords SPECIFICATION and THEORY can be considered as synonyms.

```
THEORY Problem
─────────────────────────────────────────────────────────────
  SORT domain range
─────────────────────────────────────────────────────────────
  IMPORT Set[range] ONLY set
─────────────────────────────────────────────────────────────
  FUN f : domain → set[range]
  FUN pre : domain → bool
  FUN post : domain → range → bool
─────────────────────────────────────────────────────────────
  AXM pre(x) ⇒ f(x) == {post(x)}
```

Figure 6: A theory of problems

Another elementary theory is `Enumeration-Theory` described in Section 4. Actually it is a family of theories that formalize the structure necessary for enumerating sets.

On the basis of such elementary theories we can build up more interesting theories, such as global search theory. And on top of these we may then formulate standard implementations. This way we can provide a library of standard solutions for various classes of problems.

EXAMPLE 2 *As illustrated in Figure 7, the library contains a diagram that expresses the following fact: The theory* `Problem` *is based on some range type* `R`*. When this type is an instance of the enumeration theory* `ET[R]`*, then we actually have a* global-search problem*, defined by the theory* `GS-Pr`*. For this kind of theory we possess a standard implementation* `GS-Imp`*. Moreover, the library contains the fact that the type* `Map` *is an instance of the enumeration theory* `ET[map]`*.*

When we are confronted with a concrete programming task — such as the transportation scheduling introduced above — we may analyze whether it is an instance of the `Problem` theory, and whether

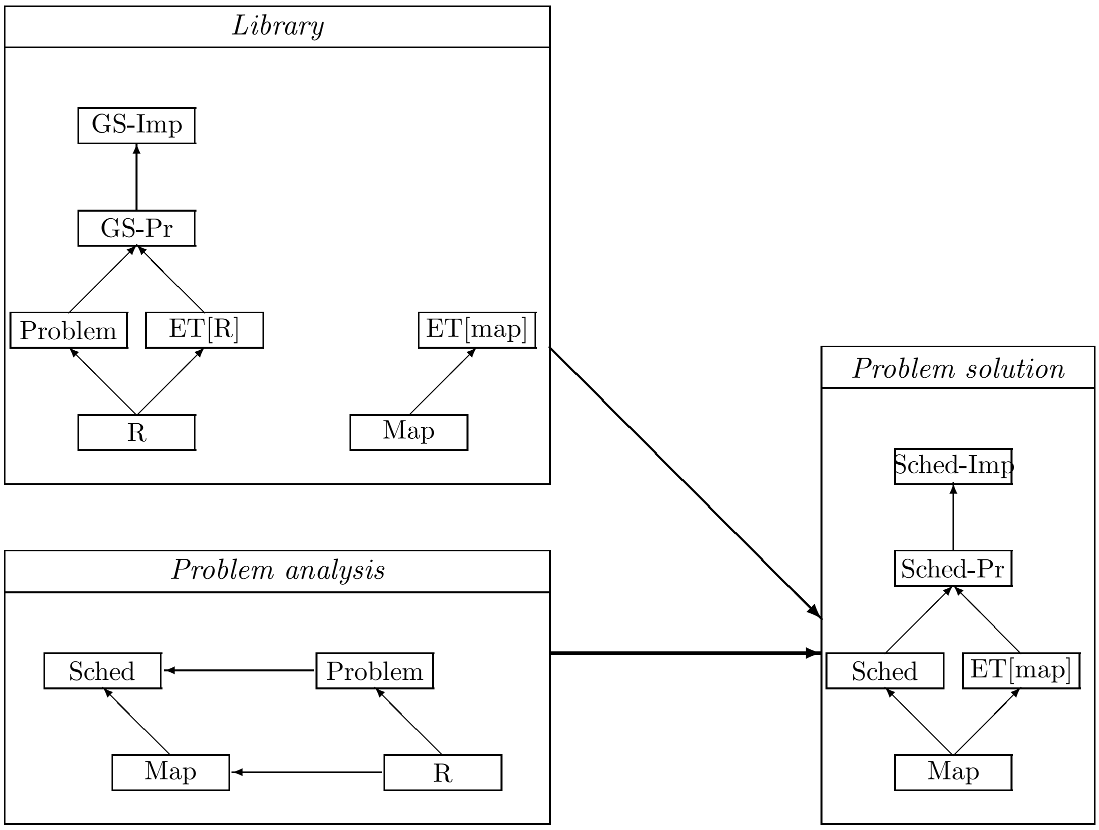


Figure 7: Applying global-search theory to the scheduling problem

its underlying range type is an instance of the enumeration theory `ET`. For the example of the scheduling task this is indeed the case (as illustrated in Figure 7). Now the proper “overlaying” of the diagram resulting from the analysis with the diagrams from the library yields the final solution for the given task (see again Figure 7). This overlaying in general requires the verification of a number of “applicability conditions”.

There are many strategic decisions that a programmer has to make; in particular, which theories shall be used and what kinds of overlaying shall take place. These decisions are often triggered by the analysis and verification of the given constraints.

Note: There is a wealth of theoretical considerations concerning the kinds of diagrams, that is, the kinds of specifications and morphisms that are used here, as well as the means for their composition. We refer the reader to the pertinent literature on SpecWare [13].

3 Search Spaces (with Constraints)

In the widest sense, *global search* means to scan a search space of potential solutions until an actual solution has been found. When describing the concepts of the pertinent algorithms, we encounter the dichotomy of

modeling by "search spaces"
versus
representation by "partial solutions".

To see what we mean by this consider the famous 8-queens problem: At some point in the algorithm we may have constructed the partial configuration $[\mathtt{q}_1 \mapsto \mathtt{(1\ ,\ 2)}\ ,\ \mathtt{q}_2 \mapsto \mathtt{(2\ ,\ 7)}]$. This partial solution represents the search space of all configurations that extend the positionings of these two queens. If we now add the positioning of a third queen $[\mathtt{q}_3 \mapsto \mathtt{(3\ ,\ 3)}]$, then this can be viewed as the extension of the partial solution or as the reduction of the remaining search space.[4]

This dichotomy is well known for a long time in recursion (removal) theory. The traditional example for its illustration is the factorial function, which can either be programmed by a loop

$$\ldots \mathtt{i} := 0;\ \mathtt{z} := 1;\ \mathbf{while}\ \mathtt{i} < \mathtt{n}\ \mathbf{do}\ \mathtt{i} := \mathtt{i} + 1;\ \mathtt{z} := \mathtt{z} * \mathtt{i}\ \mathbf{od}; \ldots$$

that proceeds upward from 0 to $\mathtt{n}$, or by the two equations

$$\mathtt{fact(i+1)} = \mathtt{(i+1)} * \mathtt{fact(i)}\ , \qquad \mathtt{fact(0)} = 1\ .$$

The latter version is directly supported in modern functional languages. Although these versions are equivalent, the latter is generally regarded as more elegant, and to be preferred.

On the basis of these observations, we feel that *modeling in terms of search spaces* is more abstract than "upward enumeration" of partial solutions and therefore better suited for describing our conceptual approach.

This abstract modeling gives us, in particular, the freedom to defer committment to specific representations. For instance, we may work with "the space of all time points in a given interval" throughout a development and commit to the representation by its two end points only towards the end of the derivation process. In the remainder of this section we elaborate on this abstract view in more detail.

3.1 A General Notion of Search Spaces

The standard representation for sets uses a **base set** and a **constraint**, formally written

$$\{\mathtt{x} \in \mathtt{S} \mid \mathtt{C(x)}\},$$

where $\mathtt{S}$ is the base set and the predicate $\mathtt{C}$ is the constraint. We will mostly use a functional-programming notation here:

$$\mathtt{S} \triangleright \mathtt{C}$$

to be read as "$\mathtt{S}$ filtered by $\mathtt{C}$". Three views are possible for this construction:

- *Semantical view*: $\mathtt{S} \triangleright \mathtt{C} = \{\mathtt{x} \in \mathtt{S} \mid \mathtt{C(x)}\}$ is a normal set, viz. that subset of $\mathtt{S}$, which comprises exactly those elements of $\mathtt{S}$ which fulfil the predicate $\mathtt{C}$.
- *Operational view*: $\mathtt{S} \triangleright \mathtt{C}$ means that we have to explicitly enumerate $\mathtt{S}$ and then send its elements through the filtering predicate $\mathtt{C}$.

[4]Technically, these two views are related by a formal program transformation known as *function inversion.*

- *Representation view*: We consider '$\triangleright$' as a type constructor (in infix notation):

TYPE `constrained-set` $==$ `_` $\triangleright$ `_` `(base :` `set[`α`]` , `constraint :` $\alpha \to$ `bool)`

Note: The empty set can be represented either as $\emptyset \triangleright$`p` with an arbitrary predicate `p`, or as `S`$\triangleright$`false` with an arbitrary base set `S`. We simply write $\emptyset$ for any of these forms.

As mentioned before, there is no need to prematurely commit to one of these views. All our considerations work with any of them. It is only towards the end of a development that we have to actually choose a representation (even though in practice some of the earlier development steps may already be geared towards an intended representation).

Note: Finding for a given set a clever separation into a base set and a constraint generally is the clue for obtaining efficiency! For instance, if the set to be represented is "the prime numbers between 40 and 50", then the best representation is the list $\langle 41, 43, 47 \rangle$*. But if the set is "the prime numbers between 1 000 and 100 000, then a better choice usually is the interval* $[1\,000..100\,000]$ — *represented simply by its two endpoints — and the predicate* `prime` *as the constraint.*

Of course, we employ all the standard operations on sets, in particular union '$\cup$' and disjoint union '$\uplus$'.

General Rules for Constrained Sets

There are a number of simple and straightforward rules that are the basis for transforming programs over constrained sets. These are collected in Figure 8 (The use of the wildcard symbol '`_`' in `( _`$\in$ `S')` is a shorthand for the lambda notation $(\lambda$`x . x` $\in$ `S'`$)$).

Note: The last law $\mathtt{CS}_7$ is, of course, a triviality. But in practice it can effect considerable gains in efficiency when the *representations* of `C` and `C'` are well-chosen.

SPECIFICATION `Constraint-Rules`

THM $\mathtt{CS}_1 : (\mathtt{S}_1 \cup \mathtt{S}_2) \triangleright \mathtt{C} = (\mathtt{S}_1 \triangleright \mathtt{C}) \cup (\mathtt{S}_2 \triangleright \mathtt{C})$
THM $\mathtt{CS}_2 : (\forall \mathtt{x} \in \mathtt{S} \,.\, \neg\, \mathtt{C(x)}) \Rightarrow \mathtt{S} \triangleright \mathtt{C} = \emptyset$
THM $\mathtt{CS}_3 : \mathtt{S} \triangleright (\mathtt{C}_1 \wedge \mathtt{C}_2) = (\mathtt{S} \triangleright \mathtt{C}_1) \triangleright \mathtt{C}_2$
THM $\mathtt{CS}_4 : \mathtt{S} \triangleright (\mathtt{C}_1 \vee \mathtt{C}_2) = (\mathtt{S} \triangleright \mathtt{C}_1) \cup (\mathtt{S} \triangleright \mathtt{C}_2)$
THM $\mathtt{CS}_5 : \mathtt{S} \triangleright (\ \_ \in \mathtt{S}') = \mathtt{S} \cap \mathtt{S}'$
THM $\mathtt{CS}_6 : \mathtt{Universe}_\alpha \triangleright (\ \_ \in \mathtt{S}') = \mathtt{S}'$ -- *special case of* C_5
THM $\mathtt{CS}_7 : (\mathtt{C} \Leftrightarrow \mathtt{C}') \Rightarrow (\mathtt{S} \triangleright \mathtt{C}) = (\mathtt{S} \triangleright \mathtt{C}')$

Figure 8: Basic rules for constrained sets

In the next sections we will consider special rules for constrained sets that are customized to specific data structures. But before we do this, we will briefly review the relation of such constrained spaces to our methodology.

Getting Smaller Search Spaces

Search spaces usually are huge, maybe even infinite. There are two ways to make them smaller.

- **Reducing** a search space means to eliminate elements from the base set which are known to violate the constraint C (see Figure 9). This means

Find a set $\mathtt{S'}$ such that $\mathtt{S} \rhd \mathtt{C} = \mathtt{S'} \rhd \mathtt{C}$

Frequently it is also possible to simultaneously pass on to a simplifed constraint:

Find a set $\mathtt{S'}$ and a constraint $\mathtt{C'}$ such that $\mathtt{S} \rhd \mathtt{C} = \mathtt{S'} \rhd \mathtt{C'}$

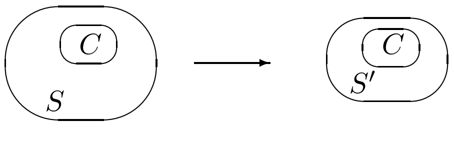


Figure 9: Reducing a search space

- **Splitting** a search space means to divide it into subspaces that are then to be searched recursively (see Figure 10). This means

  Find sets $\mathtt{S_1}$, . . . , $\mathtt{S_k}$ such that $\mathtt{S} \rhd \mathtt{C} = (\mathtt{S_1} \rhd \mathtt{C}) \cup \ldots \cup (\mathtt{S_k} \rhd \mathtt{C})$

  Frequently it is also possible to simultaneously pass on to simplifed constraints:

  Find sets $\mathtt{S_1}$, . . . , $\mathtt{S_k}$ and constraints $\mathtt{C_1}$, . . . , $\mathtt{C_k}$ such that $\mathtt{S} \rhd \mathtt{C} = (\mathtt{S_1} \rhd \mathtt{C_1}) \cup \ldots \cup (\mathtt{S_k} \rhd \mathtt{C_k})$

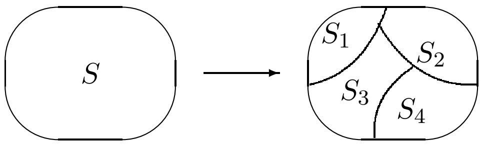


Figure 10: Splitting a search space

  Notes:

  – The splitting often enables new reductions for the subspaces.
  – Reduction can obviously be viewed as splitting into two subspaces, one of which only contains infeasible elements and therefore can be pruned away. However, there is a more useful view: reduction corresponds to the effect of adding a *conjunction* of constraints, and splitting corresponds to the effect of adding a *disjunction* of constraints.
  – Even though the subspaces need not be disjoint, this property is desirable for reasons of efficiency. If they are not, then memoization techniques may still help to obtain sufficiently fast algorithms.

From a *methodological point of view* **reduction** is generally good, because it makes the problem smaller and thus subsequent computations more efficient. By contrast, **splitting** is bad, because it generally introduces backtracking and thus exponential growth of computation costs. Therefore we have as a rule of thumb: *Do as much reduction as possible before any splitting takes place.*

Remark: As mentioned earlier, there is a duality between "search spaces" on the one side and "partial solutions" on the other side. Hence, splittings and reductions of search spaces are dual to enumerations and extensions of their representations. Therefore we keep to the more common terminology of "enumeration theories" even though we actually work with splittings and reductions of search spaces.

3.2 Spaces of Pairs (Enumeration Theories for Pairs)

Sets of pairs are built using the classical construct of the direct product $\mathtt{A} \otimes \mathtt{B}$. This operator, moreover, distributes nicely over set union. This is described in the specification `Pair-Spaces` in Figure 11.

SPECIFICATION $\mathtt{Pair\text{-}Spaces}[\alpha, \beta]$

IMPORT $\mathtt{Pair}[\alpha, \beta]$
$\mathtt{Set}[\alpha]$
$\mathtt{Set}[\beta]$

FUN $\_ \otimes \_ : \mathtt{set}[\alpha] \times \mathtt{set}[\beta] \to \mathtt{set}[\mathtt{pair}[\alpha, \beta]]$

AXM $\langle a, b\rangle \in A \otimes B \Leftrightarrow a \in A \wedge b \in B$
THM $(A \cup B) \otimes C = (A \otimes C) \cup (B \otimes C)$
THM $A \otimes (B \cup C) = (A \otimes B) \cup (A \otimes C)$
...

Figure 11: Forming spaces of pairs

It should be noted that we have at least two possible representations for the operator '$A \otimes B$'.

- *Operational view*: It represents a function that actually enumerates the set of all pairs over the given sets `A` and `B`.
- *Representation view*: We consider '$\otimes$' (in analogy to '$\triangleright$' above) as a type constructor.

Usually it will be reasonable to start a development using the type-constructor view and to change at some point in time to the enumeration-function view. (This can also be seen in our treatment of the scheduling problem below.)
In connection with constraints we obtain similarly simple properties, as is illustrated in Figure 12. Note that they hold, of course, both for the operational view and the constructor view.

SPECIFICATION $\mathtt{Pair\text{-}Constraints}[\alpha, \beta]$

IMPORT $\mathtt{Pair\text{-}Spaces}[\alpha, \beta]$

THM $(A \otimes B) \triangleright (C \circ \pi_1) = (A \triangleright C) \otimes B$
THM $(A \otimes B) \triangleright (C \circ \pi_2) = A \otimes (B \triangleright C)$

Figure 12: Constraining spaces of pairs

Note that these properties entail as special cases also laws such as

$$(A \otimes B) \triangleright ((\_ \in A') \circ \pi_1) = (A \cap A') \otimes B$$
$$((\{a\} \uplus A) \otimes B) \triangleright (C \circ \pi_1) = (A \otimes B) \triangleright (C \circ \pi_1) \qquad \text{IF } \neg\, C(a)$$
$$((\{a\} \uplus A) \otimes B) \triangleright (C \circ \pi_1) = (\{a\} \otimes B) \uplus ((A \otimes B) \triangleright (C \circ \pi_1)) \qquad \text{IF } C(a)$$

As we will see later on, such laws provide the basis for inductive enumeration algorithms.

3.3 Spaces of Maps (Enumeration Theories for Maps)

Maps are a very rich structure. Therefore we obtain a much greater variety of operations and properties for the creation, splitting, and constraining of spaces.

Specification of Map Spaces

Our task is to describe search spaces consisting of maps. We do this by introducing a number of functions in the specification `Map-Spaces` in Figure 13. Note that these functions in general produce infinite sets.

- The function $\llbracket \mathtt{A} \to \mathtt{B} \rrbracket$ yields the set of all maps from the domain `A` into the range `B`.
- Similarly, the function $\llbracket \mathtt{A} \hookrightarrow \mathtt{B} \rrbracket$ yields the set of all *total* maps from the domain `A` into the range `B`. (Most of the axioms apply to both kinds of mappings. But we only list the variants for the partial mappings here.)
- It will be convenient to lift the $\boxplus$ operator to sets of maps: let $\mathcal{M}_1 \boxplus \mathcal{M}_2$ take two sets of maps and composes every map from $\mathcal{M}_1$ with every map from $\mathcal{M}_2$.

```
SPECIFICATION Map-Spaces[α , β]
  IMPORT Map[α , β]
         Set[α]
         Set[β]

  FUN ⟦ _→_ ⟧ : set[α] × set[β] → set[map[α , β]]                 -- partial maps
  FUN ⟦ _↪_ ⟧ : set[α] × set[β] → set[map[α , β]]                 -- total maps
  FUN _ ⊞ _ : set[map[α , β]] × set[map[α , β]] → set[map[α , β]]

  AXM M ∈ ⟦A → B⟧ ⇔ domain(M) ⊆ A ∧ range(M) ⊆ B
  AXM M ∈ ⟦A ↪ B⟧ ⇔ domain(M) = A ∧ range(M) ⊆ B
  AXM M ∈ (𝓜_1 ⊞ 𝓜_2) ⇔ (∃ A ∈ 𝓜_1 , B ∈ 𝓜_2 . M = A ⊞ B)
  THM ⟦A_1 ⊎ A_2 → B⟧ = ⟦A_1 → B⟧ ⊞ ⟦A_2 → B⟧
  THM ⟦{a} → B_1 ∪ B_2⟧ = ⟦{a} → B_1⟧ ∪ ⟦{a} → B_2⟧
  THM (𝓜_1 ∪ 𝓜_2) ⊞ 𝓜_3 = (𝓜_1 ⊞ 𝓜_3) ∪ (𝓜_2 ⊞ 𝓜_3)
  AXM Universe_map[α,β] = ⟦Universe_α → Universe_β⟧
      ...
```

Figure 13: Forming spaces of maps

Notational conventions: To ease readability we introduce the following conventions:

- Small letters `a` , `b` , ... stand for elements and also for the singleton sets consisting of that element only. Capital letters `A` , `B` , ... , `M` , ... stand for sets and maps (of elements). Caligraphic letters $\mathcal{M}$, ... stand for spaces, that is, for sets of maps.
- Hence we can e.g. write $\llbracket \mathtt{a} \hookrightarrow \mathtt{B} \rrbracket$ as a shorthand for the set $\llbracket \{\mathtt{a}\} \hookrightarrow \mathtt{B} \rrbracket$ of total maps from the one-element set $\{\mathtt{a}\}$ to the set `B`.

As to the representation, we can again use several views:

- *Operational view*: Given two sets `A` and `B` we actually enumerate the set of all maps from `A` to `B`.
- *Constructor view*: We consider (in analogy to '$\triangleright$' and '$\otimes$') the operator '$\llbracket \_ \hookrightarrow \_ \rrbracket$' as a type constructor.
- *Hybrid view*: Given a domain $\mathtt{A} = \{\mathtt{a}_1 , \ldots , \mathtt{a_n}\}$ we can represent the space of mappings $\llbracket \mathtt{a}_1 \hookrightarrow \mathtt{B}_1 \rrbracket \boxplus \ldots \boxplus \llbracket \mathtt{a_n} \hookrightarrow \mathtt{B_n} \rrbracket$ as a single set-valued map $[\mathtt{a}_1 \mapsto \mathtt{B}_1 , \ldots , \mathtt{a_n} \mapsto \mathtt{B_n}]$.

Constraint Rules for Map Spaces

There are very close relationships between the above space descriptors and the constraining predicates for maps introduced in Figure 4. These relationships are collected in the specification `Map-Constraints` in Figure 14.

Note that all properties (except, of course, for `A1`) hold analogously for total maps. Moreover, property `A4` holds analogously for `pointwise-on-domain` and for `pointwise-on-range`. Similarly, `A5` holds also for `pairwise-on-domain` and for `pairwise-on-range`.

SPECIFICATION `Map-Constraints`$[\alpha , \beta]$

IMPORT `Map-Spaces`$[\alpha , \beta]$
`Map-Predicates`$[\alpha , \beta]$

THM A1 : $\llbracket \mathtt{A} \nrightarrow \mathtt{B} \rrbracket \triangleright \mathtt{pointwise\text{-}on\text{-}domain(C)} == \llbracket (\mathtt{A} \triangleright \mathtt{C}) \nrightarrow \mathtt{B} \rrbracket$
THM A2 : $\llbracket \mathtt{A} \nrightarrow \mathtt{B} \rrbracket \triangleright \mathtt{pointwise\text{-}on\text{-}range(C)} == \llbracket \mathtt{A} \nrightarrow (\mathtt{B} \triangleright \mathtt{C}) \rrbracket$
THM A3 : $\llbracket \mathtt{a} \nrightarrow \mathtt{B} \rrbracket \triangleright \mathtt{pointwise(C)} == \llbracket \mathtt{a} \nrightarrow \mathtt{B} \triangleright \mathtt{C(a,\_)} \rrbracket$
THM A4 : $(\mathcal{M}_1 \boxplus \mathcal{M}_2) \triangleright \mathtt{pointwise(C)} == (\mathcal{M}_1 \triangleright \mathtt{pointwise(C)}) \boxplus (\mathcal{M}_2 \triangleright \mathtt{pointwise(C)})$
...
THM A5 : $(\mathcal{M}_1 \boxplus \mathcal{M}_2) \triangleright \mathtt{pairwise(p)} == ((\mathcal{M}_1 \triangleright \mathtt{pairwise(C)}) \boxplus \mathcal{M}_2) \triangleright \mathtt{pairwise(C)}$
...
THM A6 : $\mathtt{S} \subseteq \mathtt{A} \Rightarrow \llbracket \mathtt{A} \nrightarrow \mathtt{B} \rrbracket \triangleright (\mathtt{domain}\ \_ = \mathtt{S}) = \llbracket \mathtt{S} \hookrightarrow \mathtt{B} \rrbracket$

Figure 14: Constraint propagation for maps

These properties actually enable a promotion *of constraints and thus a reduction of search spaces, which can dramatically improve the efficiency of the resulting algorithms. Hence, the theorems in* `Map − Constraints` *are the foundation of our approach to scheduling.*

3.4 Application to the Scheduling Problem

The scheduling task is defined in Figure 5 as `Scheduling(Cargo)` = {`Post(Cargo)`}. This means that we have as our initial search space the "universe" of all maps from movements to trips. To this universe we then apply the postcondition as constraining filter.

$$\begin{aligned}
&\texttt{Scheduling(Cargo)}\\
&= \{\texttt{Post(Cargo)}\}\\
&= \texttt{Universe}_{\texttt{map[movement,trip]}} \triangleright \texttt{Post(Cargo)}\\
&= \texttt{Universe}_{\texttt{map[movement,trip]}} \triangleright (\texttt{Complete(Cargo)} \wedge \texttt{Fleet} \wedge \texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})\\
&= [\![\texttt{Universe}_{\texttt{movement}} \rightarrow \texttt{Universe}_{\texttt{trip}}]\!] \triangleright (\texttt{Complete(Cargo)} \wedge \texttt{Fleet} \wedge \texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})
\end{aligned}$$

The first one of the constraining predicates, viz. `Complete`, is defined as[5]

```
FUN  Complete : set[movement] → schedule → bool
AXM  Complete(Cargo)(sched) == (domain(sched) = Cargo)
```

Therefore this constraint can be eliminated by virtue of the property `A6`.

$$\begin{aligned}
&\texttt{Scheduling(Cargo)}\\
&= [\![\texttt{Universe}_{\texttt{movement}} \rightarrow \texttt{Universe}_{\texttt{trip}}]\!] \triangleright (\texttt{Complete(Cargo)} \wedge \texttt{Fleet} \wedge \texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})\\
&= [\![\texttt{Cargo} \hookrightarrow \texttt{Universe}_{\texttt{trip}}]\!] \triangleright (\texttt{Fleet} \wedge \texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})
\end{aligned}$$

The second predicate, viz. `Fleet`, is defined as follows. (Note the — harmless — overloading of the identifier `Fleet`.)

```
FUN  Fleet : schedule → bool
FUN  fleet : trip → bool
AXM  Fleet(Sched) == pointwise-on-range(fleet)(Sched)
AXM  fleet(trip) == vessel(trip) ∈ Fleet
```

Therefore this constraint can be eliminated by virtue of the property `A2`. We also apply the simple rule for constraining pairs mentioned in Figure 12.

$$\begin{aligned}
&\texttt{Scheduling(Cargo)}\\
&= [\![\texttt{Cargo} \hookrightarrow \texttt{Universe}_{\texttt{trip}}]\!] \triangleright (\texttt{Fleet} \wedge \texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})\\
&= [\![\texttt{Cargo} \hookrightarrow (\texttt{Universe}_{\texttt{trip}} \triangleright \texttt{fleet})]\!] \triangleright (\texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})\\
&= [\![\texttt{Cargo} \hookrightarrow (\texttt{Universe}_{\texttt{vessel}} \otimes \texttt{Universe}_{\texttt{time}}) \triangleright \texttt{fleet}]\!] \triangleright (\texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})\\
&= [\![\texttt{Cargo} \hookrightarrow (\texttt{Fleet} \otimes \texttt{Universe}_{\texttt{time}})]\!] \triangleright (\texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})\\
&= [\![\texttt{Cargo} \hookrightarrow \texttt{Trips}]\!] \triangleright (\texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})\\
&\quad \text{WHERE}\\
&\quad \texttt{Trips} = (\texttt{Fleet} \otimes \texttt{Universe}_{\texttt{time}})
\end{aligned}$$

The third predicate, viz. `Fit`, is defined as follows.

```
FUN  Fit : schedule → bool
FUN  fit : movement × trip → bool
AXM  Fit(sched) == pointwise(fit)(sched)
AXM  fit(movement , trip) == start(trip) ∈ window(movement)
```

Therefore this constraint can be eliminated by virtue of the property `A3` and `A4`.

[5]The full specification of all five constraining predicates is listed in Figure 16 in the Appendix.

$$\begin{aligned}
&\texttt{Scheduling(Cargo)}\\
&= [\![\texttt{Cargo} \hookrightarrow \texttt{Trips}]\!] \triangleright (\texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})\\
&= ([\![\texttt{c}_1 \hookrightarrow \texttt{Trips}]\!] \boxplus \ldots \boxplus [\![\texttt{c}_n \hookrightarrow \texttt{Trips}]\!]) \triangleright (\texttt{Fit} \wedge \texttt{Sep} \wedge \texttt{Cap})\\
&= ([\![\texttt{c}_1 \hookrightarrow \texttt{Trips}_1]\!] \boxplus \ldots \boxplus [\![\texttt{c}_n \hookrightarrow \texttt{Trips}_n]\!]) \triangleright (\texttt{Sep} \wedge \texttt{Cap})\\
&\quad \text{WHERE}\\
&\quad \texttt{Trips}_i == \texttt{Trips} \triangleright \texttt{fit}(\texttt{c}_i, \_) = \texttt{Fleet} \otimes \texttt{Window}(\texttt{c}_i)
\end{aligned}$$

Of course, the use of the ellipses '...' shall just illustrate here the effect of the computation. Technically, we obtain the following recursion property:

$$\begin{aligned}
&[\![(\texttt{Cargo} \uplus \texttt{c}) \hookrightarrow \texttt{Trips}]\!] \triangleright \texttt{Fit}\\
&= ([\![\texttt{Cargo} \hookrightarrow \texttt{Trips}]\!] \boxplus [\![\texttt{c} \hookrightarrow \texttt{Trips}]\!]) \triangleright \texttt{Fit}\\
&= ([\![\texttt{Cargo} \hookrightarrow \texttt{Trips}]\!] \triangleright \texttt{Fit}) \boxplus ([\![\texttt{c} \hookrightarrow \texttt{Trips}]\!] \triangleright \texttt{Fit})\\
&= ([\![\texttt{Cargo} \hookrightarrow \texttt{Trips}]\!] \triangleright \texttt{Fit}) \boxplus [\![\texttt{c} \hookrightarrow (\texttt{Trips} \triangleright \texttt{fit}(\texttt{c}, \_))]\!]\\
&= ([\![\texttt{Cargo} \hookrightarrow \texttt{Trips}]\!] \triangleright \texttt{Fit}) \boxplus [\![\texttt{c} \hookrightarrow (\texttt{Fleet} \otimes \texttt{Universe}_{\texttt{time}}) \triangleright \texttt{fit}(\texttt{c}, \_)]\!]\\
&= ([\![\texttt{Cargo} \hookrightarrow \texttt{Trips}]\!] \triangleright \texttt{Fit}) \boxplus [\![\texttt{c} \hookrightarrow \texttt{Fleet} \otimes \texttt{Window(c)}]\!]
\end{aligned}$$

This concludes the easy part. The remaining two filters require the essential work.

3.5 Fixpoint Calculation for Constraints

The filter `Sep` is defined as follows.

```
FUN Sep : schedule → bool
FUN sep : trip × trip → bool
AXM Sep(sched) == pairwise-on-range(sep)(sched)
AXM sep(trip₁ , trip₂) ==
     (vessel(trip₁) = vessel(trip₂) ⇒  start(trip₁) = start(trip₂)
                                     ∨ start(trip₁) + roundtrip ≤ start(trip₂)
                                     ∨ start(trip₁) − roundtrip ≥ start(trip₂))
```

This filter exhibits two kinds of complications:

- First, due to the `pairwise-on-range` filter it establishes dependencies across multiple entries of the map.
- Second, the `sep` predicate corresponds to a system of inequations which are solvable by fixpoint iteration.

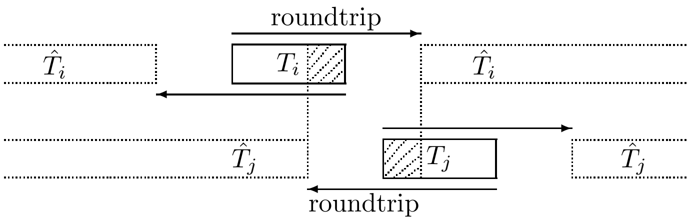
To get a better understanding of the pertinent formal definitions, let us consider the situation informally first. As far as the constraint `Sep` is concerned, the relevant situations are of the following kind:

$$[\ldots, \texttt{m}_i \mapsto \langle \texttt{v}, \texttt{t}_i \rangle, \ldots, \texttt{m}_j \mapsto \langle \texttt{v}, \texttt{t}_j \rangle, \ldots]$$

where $\texttt{m}_i$ and $\texttt{m}_j$ are movements, `v` is a vessel, and $\texttt{t}_i$ and $\texttt{t}_j$ are start times. Since both movements shall go on the same vessel, they have to obey the constraint (where `rt` is short for `roundtrip`)

$$(\texttt{t}_i = \texttt{t}_j) \vee (\texttt{t}_j \leq \texttt{t}_i - \texttt{rt}) \vee (\texttt{t}_i + \texttt{rt} \leq \texttt{t}_j) \ .$$

Since we are still working on the level of whole search spaces, we do not yet have individual start times $\texttt{t}_i$ and $\texttt{t}_j$, but rather whole sets (that is, intervals) $\texttt{T}_i$ and $\texttt{T}_j$ of possible start times. So the `Sep` constraint actually becomes a constraint on intervals:



As illustrated in this diagram each of the windows $\mathtt{T_i}$ blocks an extended window from other trips, due to the round-trip duration. If we denote the area *outside* this extended window by $\hat{\mathtt{T}}_\mathtt{i}$, then our constraint becomes

$$(\mathtt{T_i} = \mathtt{T_j}) \vee (\mathtt{T_i} \subseteq \hat{\mathtt{T}}_\mathtt{j} \wedge \mathtt{T_j} \subseteq \hat{\mathtt{T}}_\mathtt{i})$$

Note: The subset relation can be easily expressed in terms of basic arithmetic. Let e.g. A be left of B. Then $(\mathtt{B} \subseteq \hat{\mathtt{A}})$ stands for $\mathtt{B} \subseteq \mathtt{[max(left(B)\ ,\ left(A)} + \mathtt{rt)\ ..\ right(B)]}$.

The disjunctive nature of the Sep constraint leads to difficulties, since disjunctions correspond to splitting of search spaces. Fortunately, it is possible to simplify Sep considerably by exploiting transitivity. Supposing that we can determine when one trip immediately predeces another (via the immediate-predecessor relation), then sep can be rewritten as a conditional inequality:

```
sep(trip$_1$ , trip$_2$) ==
      (vessel(trip$_1$) = vessel(trip$_2$)
      $\wedge$ immediate-predecessor(trip$_1$ , trip$_2$)
      $\Longrightarrow$ start(trip$_1$) + roundtrip $\leq$ start(trip$_2$)) .
```

This form implies the original disjunctive form by transitivity. Semantically we obtain the properties $\hat{\mathtt{T}}_{\mathtt{i}+1} \subseteq \hat{\mathtt{T}}_\mathtt{i}$ (where the $\hat{\mathtt{T}}$ now only denote the upper areas).

On the basis of these considerations we obtain the following situation, where the sets $\mathtt{Win_i}$ are the solutions (by way of fixpoint iteration) of the derived system of inequations sketched above.

$$\begin{aligned}
&\mathtt{Scheduling(Cargo)}\\
&= ([\![\mathtt{c_1} \hookrightarrow \mathtt{Trips_1}]\!] \boxplus \ldots \boxplus [\![\mathtt{c_n} \hookrightarrow \mathtt{Trips_n}]\!]) \rhd (\mathtt{Sep} \wedge \mathtt{Cap})\\
&= ([\![\mathtt{c_1} \hookrightarrow \mathtt{Trips'_1}]\!] \boxplus \ldots \boxplus [\![\mathtt{c_n} \hookrightarrow \mathtt{Trips'_n}]\!]) \rhd \mathtt{Cap}\\
&\quad \text{WHERE}\\
&\quad \mathtt{Trips_i} == \mathtt{Fleet} \otimes \mathtt{Window(c_i)}\\
&\quad \mathtt{Trips'_i} == \mathtt{Fleet} \otimes \mathtt{Win_i}
\end{aligned}$$

It should be noted that the hybrid representation mentioned in Section 3.3 is particularly useful here. In that case we deal with the single, set-valued map

$$[\mathtt{c_1} \mapsto \mathtt{Trips'_1}\ , \ldots ,\ \mathtt{c_n} \mapsto \mathtt{Trips'_n}]$$

Let us briefly also consider the pertinent inductive equation in the place of the '. . .':

$$\begin{aligned}
&[\![\mathtt{Cargo} \uplus \mathtt{c} \hookrightarrow \mathtt{Trips}]\!] \rhd \mathtt{Sep}\\
&= ([\![\mathtt{Cargo} \hookrightarrow \mathtt{Trips}]\!] \boxplus [\![\mathtt{c} \hookrightarrow \mathtt{Trips}]\!]) \rhd \mathtt{Sep}\\
&= (([\![\mathtt{Cargo} \hookrightarrow \mathtt{Trips}]\!] \rhd \mathtt{Sep}) \boxplus [\![\mathtt{c} \hookrightarrow \mathtt{Trips}]\!]) \rhd \mathtt{Sep}\\
&= (\mathcal{M} \boxplus [\![\mathtt{c} \hookrightarrow \mathtt{Trips}]\!]) \rhd \mathtt{Sep}\\
&\quad \text{WHERE}\\
&\quad \mathcal{M} == [\![\mathtt{Cargo} \hookrightarrow \mathtt{Trips}]\!] \rhd \mathtt{Sep}
\end{aligned}$$

It is important that each map in $\mathcal{M}$ already fulfils the constraint Sep. So, when adding an assignment $\texttt{[c} \mapsto \langle\texttt{vessel},\texttt{Window}\rangle\texttt{]}$ to some map $\texttt{M} \in \mathcal{M}$ we "only" have to incrementally correct the existing windows in M with respect to Window. As is known from the literature – see e.g. [6] – such an incremental correction is possible in mathematical structures such as ours here.

In more detail

To continue the example, we first focus on the simplified situation

$$\begin{aligned}&(\llbracket a_1 \hookrightarrow B_1\rrbracket \boxplus \ldots \boxplus \llbracket a_n \hookrightarrow B_n\rrbracket) \triangleright \texttt{pairwise-on-range(C)}\\ &= \llbracket a_1 \hookrightarrow B'_1\rrbracket \boxplus \ldots \boxplus \llbracket a_n \hookrightarrow B'_n\rrbracket\end{aligned}$$

where the predicate C is such that the filter entails for any two sets B'_i , B'_j an inequation of the form

$$B'_i \subseteq h_{i,j}(B'_j)$$

where $h_{i,j}$ is a monotone function. Such a system of inequations can be solved via fixpoint iteration by starting from the sets $B'_i = B_i$ and calculating iteratively $B'_i := B'_i \cap h_{i,j}(B'_j)$ until all sets stabilize. As mentioned above, this can be done incrementally when the maps are built up using the construction

$$(\mathcal{M} \boxplus \llbracket a \hookrightarrow B\rrbracket) \triangleright \texttt{pairwise-on-range(C)} = \mathcal{M}' \boxplus \llbracket a \hookrightarrow B'\rrbracket$$

KIDS uses a theorem-prover to support the derivation of inequalities from constraints. Often the derivation is simple and automatic; sometimes the user must supply lemmas to help the prover. See [12] for more details on the derivation of propagation code.

3.6 Constraints on Inverse Maps

The second truly complex constraint is Cap. It is defined as follows:

```
FUN  Cap : schedule → bool
FUN  cap : trip × set[movement] → bool
AXM  Cap(sched) == pointwise-on-inverse-map(cap)(sched)
AXM  cap(trip , Moves) == sum(size)(Moves)) ≤ capacity(vessel(trip))
```

This constraint imposes again dependencies across several map entries. However, it can be relatively easily transformed into a simple "pointwise" constraint. All we have to do is to choose a particular ("inverted") representation for our maps: A map

$$[a_1 \mapsto b_1 , \ldots , a_n \mapsto b_n]$$

can be represented in a form which associates a whole set of domain elements to every range element.

$$\llbracket A_1 \hookrightarrow b_1\rrbracket \boxplus \ldots \boxplus \llbracket A_n \hookrightarrow b_n\rrbracket$$

That is, A_i comprises all domain elements that are associated to b_i.
In this representation the separation constraint becomes a trivial pointwise constraint.

3.7 Representing Schedules

The previous two subsections have suggested refinements to the maps that we are using to represent schedules. In Section 3.5 there is a need to readily enumerate over pairs of trips that are immediate predecessors. Section 3.6 suggests the need to represent the schedule as an inverse map, collecting all the movement requirements on a given trip. These two requirements can be met by refining `schedule` into maps from vessels to sequences of trips: `map[vessel , sequence(trip')]` where `trip'` == `(start : time , load : set[Cargo])`. The sequence of trips associated with each vessel allows easy enumeration of adjacent pairs of trips, and `trip'` directly represents the load of a trip to ease the checking of the `Cap` constraint. This data structure, which is used in the KIDS derivation, is more complex to work with during the derivation (in terms of the necessary laws and inferences), but is more efficient with respect to propagation operations.

4 Operational View of Enumeration Theories

In the previous sections we have performed algebraic developments which essentially derived new equations from given specifications. Now we want to show that these developments indeed lead to constructive algorithms.

4.1 Set-theoretic Equations as Recursive Programs

We claim that the equational specifications of our various functions actually entail operational implementations as a borderline case. To illustrate this claim, let us consider the following equalities that are directly derivable from the specification `Map-Spaces`. Given a set $B = \{b_1, \ldots, b_n\}$, we obtain the equality

$$\begin{aligned}&[\![a \hookrightarrow b_1 \uplus B']\!] \boxplus \mathcal{M}\\ &= ([\![a \hookrightarrow b_1]\!] \uplus [\![a \hookrightarrow B']\!]) \boxplus \mathcal{M}\\ &= (\{[a \mapsto b_1]\} \boxplus \mathcal{M}) \uplus ([\![a \hookrightarrow B']\!] \boxplus \mathcal{M})\end{aligned}$$

But we can as well deduce another equation:

$$\begin{aligned}&[\![a \hookrightarrow B]\!] \boxplus \mathcal{M}\\ &= \{[a \mapsto b_1], \ldots, [a \mapsto b_n]\} \boxplus \mathcal{M}\end{aligned}$$

If we now interpret '$\uplus$' as a **lazy operation** (more precisely: as non-strict in its second argument), then the first variant entails a different calculation than the second one. In the first equation we associate `a` to b_1 and then combine this map with all maps in $\mathcal{M}$, before we consider the second association of `a` to b_2, etc. By contrast, in the second form we first construct all possible associations for `a` and then combine each of them with the maps in $\mathcal{M}$.

This effect can be used in order to do "goal-directed" equational reasoning. That is, we transform certain terms into other syntactic forms which are semantically equivalent but entail — under a lazy interpretation — a different and hopefully more efficient computation.

4.2 Accumulator Transformations

For some of the subsequent considerations we have to briefly review a well-known transformation technique that sheds more light on the comments made at the beginning of Section 3. The essential prerequisite is an associative function like our composition operator '$\boxplus$' for maps. Then we can convert a recursive function of the kind

$$\texttt{f(...)} = \texttt{...M} \boxplus \texttt{f(...)...}$$

into an essentially equivalent function using an additional parameter `A` for accumulating the result

$$\texttt{f}'\texttt{(...,A)} = \texttt{...f}'\texttt{(..., A} \boxplus \texttt{M)...}$$

For the details of this transformation we refer to [1]. In our context it means that we pass from equations such as

$$[\![\texttt{a} \hookrightarrow \texttt{b}]\!] \boxplus [\![\texttt{A} \hookrightarrow \texttt{B}]\!]$$

to equations of the form

$$(\texttt{M} \boxplus [\texttt{a} \mapsto \texttt{b}]) \boxplus [\![\texttt{A} \hookrightarrow \texttt{B}]\!]$$

where we again take the liberty of applying the operator $\boxplus$ to single maps as well. This latter form shows quite clearly how the elements of our search spaces are built up incrementally – this is what we referred to as "partial solutions" in Section 3.

4.3 Strategies (Heuristics)

The effects of the above considerations can be utilized by the programmer to realize design strategies. To see this, let us consider again our scheduling example. After the first two steps we had arrived at the following version:

$$[\![\texttt{Cargo} \hookrightarrow (\texttt{Fleet} \otimes \texttt{Universe}_{\texttt{time}})]\!] \rhd \dots$$

By using the splittings $\texttt{Cargo} = \texttt{c} \uplus \texttt{Crg}$ and $\texttt{Fleet} = \texttt{v} \uplus \texttt{Flt}$ we can perform the following deductions:

$$\begin{array}{ll}
 & [\![\texttt{Cargo} \hookrightarrow (\texttt{Fleet} \otimes \texttt{Universe}_{\texttt{time}})]\!] \rhd \dots \\
\texttt{(1)} & = [\![\texttt{c} \uplus \texttt{Crg} \hookrightarrow (\texttt{Fleet} \otimes \texttt{Universe}_{\texttt{time}})]\!] \rhd \dots \\
 & = ([\![\texttt{c} \hookrightarrow (\texttt{Fleet} \otimes \texttt{Universe}_{\texttt{time}})]\!] \boxplus [\![\texttt{Crg} \hookrightarrow (\texttt{Fleet} \otimes \texttt{Universe}_{\texttt{time}})]\!]) \rhd \dots \\
 & = ([\![\texttt{c} \hookrightarrow (\texttt{Fleet} \otimes \texttt{Window(c)})]\!] \boxplus [\![\texttt{Crg} \hookrightarrow (\texttt{Fleet} \otimes \texttt{Universe}_{\texttt{time}})]\!]) \rhd \dots \\
\texttt{(2)} & = ([\![\texttt{c} \hookrightarrow (\texttt{v} \uplus \texttt{Flt} \otimes \texttt{Window(c)})]\!] \boxplus \dots) \rhd \dots \\
 & = (([\![\texttt{c} \hookrightarrow (\texttt{v} \otimes \texttt{Window(c)})]\!] \uplus [\![\texttt{c} \hookrightarrow (\texttt{Flt} \otimes \texttt{Window(c)})]\!]) \boxplus \dots) \rhd \dots \\
 & = (([\![\texttt{c} \hookrightarrow (\texttt{v} \otimes \texttt{Window(c)})]\!] \boxplus \dots) \uplus ([\![\texttt{c} \hookrightarrow (\texttt{Flt} \otimes \texttt{Window(c)})]\!] \boxplus \dots) \rhd \dots
\end{array}$$

The points marked with (1) and (2) indicate **decision points**:

- At point (1) we determine the order in which the movement requirements are handled. Based on the consideration in Section 3.5 it is advisable to choose the cargo items in ascending order of latest start times. This way the transitivity criterion mentioned in Section 3.5 is more easily met. This heuristic is trivially implemented by representing the cargo items as an ordered list such that the cargo `c` in the splitting expression $\texttt{c} \uplus \texttt{Crg}$ always is the first element in the order.

- At point (2) we have to choose a suitable vessel. However, in order to see possible decisions we first have to apply the accumulator transformation mentioned above. This means that we are actually confronted with a situation where we add an association for a new cargo item `c` to a partial schedule `M`:

$$\begin{aligned}&\ldots\\ &= (\mathtt{M} \boxplus [\![\mathtt{c} \hookrightarrow (\mathtt{v} \uplus \mathtt{Flt} \otimes \mathtt{Window(c)})]\!] \boxplus \ldots) \triangleright \ldots\end{aligned}$$

For this choice we have two options:

– Either we try to choose a vehicle `v` that already appears in `M`. This strategy will tend to generate schedules that use a relatively small number of vessels. It therefore should be chosen when the minimization of the required fleet is the objective function. This can be trivially implemented by representing `Fleet` as an ordered list such that the splitting $\mathtt{Fleet} = \mathtt{v} \uplus \mathtt{Flt}$ always refers to the same vessel `v`.

– Alternately, we try to choose vessels according to earliest availability regardless of whether they occurs in `M`. This strategy will tend to keep the overall time short because it utilizes the whole available fleet for transportation. The implementation is here slightly more intricate. But in connection with the inverse-map representation that we need for the constraint `Cap` in Section 3.6 anyway the necessary information is quite readily available.

These examples suffice to illustrate the interplay between the general development pattern and individual strategic decisions by the programmer, leading to heuristic guidance of the search process.

4.4 Enumeration Theories

The general concept of enumeration theories allows us to effectively construct new sets from given sets, provided that the given sets are effectively enumerable as well. Technically we have to distinguish various cases, in which the new set is constructed from one or two or three ... given sets. For instance, the enumeration of the set of all sequences over a given set `A` is given by a one-set enumeration theory. Our examples of pair spaces and map spaces are instances of two-set enumeration theories, so we consider such a theory here (see Figure 15).

In this specification the property `Constructive` means that the operator is e.g. the union operator '$\cup$' or '$\uplus$', or some kind of map or reduce morphism as used by R. Bird and L. Meertens [2, 4].

In our two applications we have the following instances:

- For pairs, both '$\odot_1$' and '$\odot_2$' are '$\uplus$'.
- For maps, '$\odot_1$' is '$\boxplus$', and '$\odot_2$' is '$\uplus$'.

One could actually lift the abstraction one level higher by basing the whole enumeration concept on some kind of "constructive" partial orders. However, we refrain from going into more technical details here because the overall paradigm should have become evident from the extended treatment of the scheduling problem.

5 Conclusion

The derivation of this scheduling algorithm in KIDS proceeds at a lower level and hence is more difficult to understand. It also makes use of many more rules. However, the resulting code runs orders of magnitude faster than comparable schedulers [11]. Kestrel researchers are building a new

THEORY Enumeration-Theory$[\alpha , \beta , \gamma]$

REQUIRE Enumeration-Theory$[\alpha]$
REQUIRE Enumeration-Theory$[\beta]$

IMPORT Set$[\alpha]$
IMPORT Set$[\beta]$
IMPORT Set$[\gamma]$

FUN Enum : $\texttt{set}[\alpha] \times \texttt{set}[\beta] \to \texttt{set}[\gamma]$
FUN $\odot_1$: $\texttt{set}[\gamma] \times \texttt{set}[\gamma] \to \texttt{set}[\gamma]$
FUN $\odot_2$: $\texttt{set}[\gamma] \times \texttt{set}[\gamma] \to \texttt{set}[\gamma]$

AXM $\texttt{Enum}(A_1 \uplus A_2 , B) = \texttt{Enum}(A_1 , B) \odot_1 \texttt{Enum}(A_2 , B)$
AXM $\texttt{Enum}(A , B_1 \uplus B_2) = \texttt{Enum}(A , B_1) \odot_2 \texttt{Enum}(A , B_2)$
AXM Constructive$[\odot_1]$
AXM Constructive$[\odot_2]$

Figure 15: Basic theory for constructive enumeration

sytem, called SpecWare [13], to replace Kids and we expect that derivations in SpecWare will be closer to the style and level of abstraction presented in this paper.

Acknowledgement

We thank our colleagues from Kestrel Institute, notably Richard Jüllig and Yellamraju Srinivas, for many stimulating discussions during the derivation of this paper.

References

[1] Bauer, F.L., Wössner, H. Algorithmic Language and Program Development. Springer Verlag, Berlin, 1982.

[2] Bird, R. S. Introduction to the theory of lists. In *'Logic of Programming and Calculi of Discrete Design* (1987), M. Broy, Ed., Springer-Verlag, NATO ASI Series F: Computer and Systems Sciences, Vol 36.

[3] Burstein, M., and Smith, D. ITAS: A portable interactive transportation scheduling tool using a search engine generated from formal specifications. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)* (Edinburgh, UK, May 1996).

[4] Meertens, L. Constructing a claculus of programs. In *Proceedings of the International Conference on Mathematics of Program Construction, LNCS 375*, L. van de Snepscheut, Ed. Springer-Verlag, Berlin, 1989, pp. 66–90.

[5] Didrich, K., Fett, A., Gerke, C., Grieskamp, W., Pepper, P. Opal: Design and implementation of an algebraic programming language. In: Gutknecht, J. (ed.): *Proc. Intl. Conf. on Programming Languages and System Architectures* (Zürich, March 1994), LNCS 782, Springer Verlag, Berlin, pp. 228–244.

[6] Cai, J., and Paige, R. Program derivation by fixed point computation. *Science of Computer Programming 11* (1989), 197–261.

[7] Smith, D. R. Structure and design of global search algorithms. Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987.

[8] Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.

[9] Smith, D. R., and Lowry, M. R. Algorithm theories and design tactics. *Science of Computer Programming 14*, 2-3 (October 1990), 305–321.

[10] Smith, D. R. Transformational approach to scheduling. Tech. Rep. KES.U.92.2, Kestrel Institute, November 1992.

[11] Smith, D. R., and Parra, E. A. Transformational approach to transportation scheduling. In *Proceedings of the Eighth Knowledge-Based Software Engineering Conference* (Chicago, IL, September 1993), pp. 60–68.

[12] Smith, D. R., Parra, E. A., and Westfold, S. J. Synthesis of high-performance transportation schedulers. Tech. Rep. KES.U.95.6, Kestrel Institute, March 1995.

[13] Srinivas, Y. V., and Jüllig, R. Specware$^{\mathrm{TM}}$: Formal support for composing software. in *Proceedings of the Conference on Mathematics of Program Construction*, LNCS 947, Kloster Irsee, Germany, July 1995.

6 Appendix: Full Specification of the Scheduling Problem

For the sake of completeness we present in Figure 16 the full specification `Scheduling-Basics` that defines the predicates for expressing the various constraints for our transportation scheduling example.

```
SPECIFICATION Scheduling-Basics

  IMPORT ...

  TYPE schedule == map[movement , trip]
  TYPE movement == (window : interval ,  size :  size)
  TYPE trip     == (vessel : vessel , start :  time)
  TYPE vessel   == (capacity : number , ... )
  TYPE time     == ...
  TYPE size     == ...
  FUN Fleet :  set[vessel]
  FUN roundtrip :  time

  FUN Fit :  schedule → bool
  FUN fit :  movement × trip → bool
  AXM Fit(sched)  ==  pointwise(fit)(sched)
  AXM fit(movement , trip)  ==  start(trip) ∈  window(movement)

  FUN Sep :  schedule → bool
  FUN sep :  trip × trip → bool
  AXM Sep(sched)  ==  pairwise-on-range(sep)(sched)
  AXM sep(trip₁ , trip₂)  ==  (vessel(trip₁) = vessel(trip₂) ⇒
       start(trip₁) = start(trip₂)
      ∨ start(trip₁) + roundtrip ≤ start(trip₂)
      ∨ start(trip₁) − roundtrip ≥ start(trip₂))

  FUN Cap :  schedule → bool
  FUN cap :  trip × set[movement] → bool
  AXM Cap(sched)  ==  pointwise-on-inverse-map(cap)(sched)
  AXM cap(trip , Moves)  ==  sum(size)(Moves)) ≤ capacity(vessel(trip))

  FUN Fleet :  schedule → bool
  FUN fleet :  trip → bool
  AXM Fleet(sched)  ==  pointwise-on-range(fleet)(sched)
  AXM fleet(trip)  ==  vessel(trip) ∈  Fleet

  FUN  Complete :  set[movement] × schedule → bool
  AXM Complete(Cargo , sched)  ==  domain(sched) = Cargo
```

Figure 16: Basic concepts of the scheduling problem