COMPUTING SCIENCE

How to say why (in AI4FM)

Leo Freitas, Cliff B. Jones, Andrius Velykis and Iain Whiteside

TECHNICAL REPORT SERIES

No. CS-TR-1398 October 2013

TECHNICAL REPORT SERIES

No. CS-TR-1398 October, 2013

How to say why (in AI4FM)

L. Freitas, C.B. Jones, A. Velykis, and I. Whiteside

Abstract

In the AI4FM project we have set ourselves the challenge of building a system that can learn high-level proof strategies by monitoring expert users. A typical level of ambition is users who are proving the feasibility and reification of medium-sized specifications. The purpose of this report is to provide a source document. In particular, it (a) summarises some experiments in the use of verification tools to determine how realistic the ambition is of extracting the "why" from experts' use of verification tools; and,(b) provides a revision of an earlier description of an abstract model of an AI4FM system that is linked to the case studies.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

FREITAS, L., JONES, C.B., VELYKIS, A., WHITESIDE, I.

How to Say Why (in AI4FM)
[By] L. Freitas, C. B. Jones, A. Velykis and I. Whiteside

Newcastle upon Tyne: Newcastle University: Computing Science, 2013.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1398)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1398

Abstract

In the AI4FM project we have set ourselves the challenge of building a system that can learn high-level proof strategies by monitoring expert users. A typical level of ambition is users who are proving the feasibility and reification of medium-sized specifications. The purpose of this report is to provide a source document. In particular, it (a) summarises some experiments in the use of verification tools to determine how realistic the ambition is of extracting the "why" from experts' use of verification tools; and,(b) provides a revision of an earlier description of an abstract model of an AI4FM system that is linked to the case studies.

About the authors

Leo Freitas is a lecturer in Formal Methods working on the EPSRC-funded AI4FM project at Newcastle University. Leo received his PhD in 2005 from the University of York with a thesis on 'Model Checking Circus', which combined refinement-based programming techniques with model checking and theorem proving. Leo's expertise is on theorem proving systems (e.g. Isabelle, Z/EVES, ACL2, etc.) and formal modelling (e.g. Z, VDM, Event-B), with particular interest on models of industrial-scale. Leo has also contributed extensively to the Verified Software Initiative (VSTTE).

Cliff B. Jones is currently Professor of Computing Science at Newcastle University. As well as his academic career, Cliff has spent over 20 years in industry. His 15 years in IBM saw among other things the creation –with colleagues in Vienna– of VDM which is one of the better known “formal methods”. Under Tony Hoare, Cliff wrote his doctoral thesis in two years. From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which –among other projects– was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the “mural” (Formal Method) Support Systems theorem proving assistant). He is now applying research on formal methods to wider issues of dependability. He is PI of two EPSRC-funded responsive mode projects “AI4FM” and "Taming Concurrency", CI on the Platform Grant TrAmS-2 and also leads the ICT research in the ITRC Program Grant. Cliff is a Fellow of the Royal Academy of Engineering (FREng), ACM, BCS, and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973.

Andrius Velykis is a PhD student in AI4FM project at Newcastle University, working under supervision of Prof. Cliff Jones. He is interested in understanding and learning from interactive proof. His PhD research investigates how to extract high-level proof ideas from an interactive proof - these ideas could then be reused to complete similar proofs automatically. Andrius gained his BSc in Applied Mathematics at Kaunas University of Technology, Lithuania. He continued his studies with MSc Software Engineering at the University of York, UK, where he was awarded the degree with distinction.

Iain J. Whiteside is a research associate on the EPSRC-funded AI4FM project at Newcastle University. Iain received his PhD in 2013 from the University of Edinburgh with a thesis 'Refactoring Proofs', which introduced a formal framework for reasoning about transformation on formal proof languages. In particular, his paper `Towards Formal Proof Script Refactoring' provided a mathematical understanding for refactoring in formal proof developments and won the best paper award at Conferences on Intelligent Computer Mathematics (CICM).
He has also worked at NASA Ames Research Center, developing a hierarchical structuring mechanism for Safety Cases in the AdvoCATE tool developed at Ames. He organised the 2013 CIAO workshop in Scotland, and the 5th Programming Languages for Mechanised Mathematical Systems workshop, a satellite workshop of CICM 2013 in Bath.

Suggested keywords

PROOF PROCESS
PROOF META-MODEL
PROOF STRATEGIES

How to say why (in AI4FM)

Leo Freitas, Cliff B. Jones, Andrius Velykis, Iain Whiteside

School of Computing Science, Newcastle University
{name.surname}@newcastle.ac.uk

October 30, 2013

Abstract

In the AI4FM project we have set ourselves the challenge of building a system that can learn high-level proof strategies by monitoring expert users. A typical level of ambition is users who are proving the feasibility and reification of medium-sized specifications. The purpose of this report is to provide a source document. In particular, it (a) summarises some experiments in the use of verification tools to determine how realistic the ambition is of extracting the "why" from experts' use of verification tools; and, (b) provides a revision of an earlier description of an abstract model of an AI4FM system that is linked to the case studies.

Contents

Chapter 1

Introduction

In a UK-funded project known as AI4FM[1] we have set ourselves the challenge of learning proof strategies from experts. The challenge is discussed in several earlier publications including [JFV13].

The purpose of this report is to provide source material from an experiment in the use of verification tools to determine how realistic the ambition is of extracting the "why" from experts' use of verification tools. That is, the underlying intent behind certain decisions in both modelling, proof strategies, the way to "phrase" lemmas, *etc.* It is useful to think of the task we face in deducing/re-using (high level) strategies by comparing it to the task of designing a "programming language" — we find it better to design a language from its state. Our hypothesis for the project is:

> *Enough information-extraction can be automated from a mechanical proof that future proofs of examples of the same class can have increased automation*

One crucial point: the importance of starting the analysis of what the user (expert) is doing top down — this is the key to getting an appropriate "parse" of the expert's steps. Looking at the proof steps from a finished proof script is a much harder way of understanding what is going on. Our hope was to enable through this process transference of proof strategies between problems to the point of getting full automation. Although we still believe it is achievable, we are still a way off, hence the use of "increased automation" instead of "full automation" in the hypothesis.

In this report, a non-trivial example is used. We believe that too small an example is unlikely to clarify the issues with high-level proof strategies; on the other hand, genuine industrial examples are just too large for consideration. We also present general lessons from such larger proof exercises, such as [DEP12a, FW08, DEP12b, FW09, Sch12, JOW06]. We are clear that AI4FM will only achieve its objectives if it can work for examples as large as those used by Schmalz in his admirable engineering comparisons in [Sch12], by other examples met in the DEPLOY project[2] or in [FW08, Fre04, FW09]. This is, however, for the future. The "Heap" example here offers just enough challenges to illustrate key points in our approach to gathering information from an expert's proof (see Chapters 2 and 4 for more detail).

In [JFV13], we describe the principles and processes behind what we believe must be captured during an expert's proof in order to be able to replay the ideas in related contexts. In the current report, we describe both the thought processes behind a modelling exercise, as well as the "proof engineering" principles to get there. We outline key stages of the development process in our description, including mistaken paths and their correction. We also summarise principles from larger examples worked so far.

[1] See `http://www.ai4fm.org`

[2] See `http://www.deploy-project.eu/`

Before a stated conjecture is proved, initial failures are common and have various sources. Reasons include, but are not limited to, mistaken understanding of mathematical notation by engineers, misinterpretation of requirements and unnecessarily complex modelling decisions, etc. Proof experts can fail initially as they are not necessarily familiar with the problem domain. In an industrial setting where hundreds of (structurally similar) proof obligations emerge, this is a serious problem. By prescriptively capturing the intent behind what an expert/engineer does and –more importantly– how does one recover from failure, our aim is to reduce the amount of effort involved in discharging remaining proof obligations, once a proof is finished. That is, by analysing the way experts (fail and) produce proofs, we hope to transfer some proof ideas from one problem to another through a set of expert proofs.

Our approach was to mechanise models using two different theorem provers in order to test our hypothesis that proof intent, and sometimes even strategies, are transferable between problems of similar shape. This is less difficult to identify across problems within the same prover, yet it also transfers between provers in some cases. Obviously, different provers have different strengths and ways of interaction. We focus on Isabelle (see Chapter 5), and discuss additional proof efforts in Z/EVES in an appendix (see Appendix G and Chapter 4).

The heap example comes from [JS90, Chapter 7] and uses VDM (e.g. [Jon90]), which is similar to other model-oriented specification languages used in industry, such as Z [WD96] and (Event-)B [Abr96, Abr10]. Perhaps the least uniform decision between such formal languages is how they handle undefined terms that arise from the application of partial functions. VDM uses the "Logic of Partial Functions" [BCJ84, JLS12], which can be thought of as a three-valued logic.

VDM generates proof obligations for well-formedness (i.e. specifications denote: functions are within their domain and unique existential quantification are checked), feasibility of state operations (i.e. operation preconditions are strong enough to ensure that postconditions can be satisfied) and data reification proof obligations (PO) (i.e. that changes in data representations to add extra detail respect previous design decisions).

In our experiment, we first typeset models and all its layers of refinement using the VDM Overture Tools[3]. This enabled us to identify a few minor errors in typing (e.g. sequence types used as sets) and other minor syntactical issues. Overture generates well-formedness and feasibility proof obligations but, unfortunately, no refinement POs. Moreover, there is no theorem proving support for VDM to our knowledge.

To discharge proof obligations, we used two theorem provers: Isabelle [Pau94, NPW02a, WWW13] and Z/EVES [Saa99], the former is a well-known general purpose theorem prover, whereas the latter is a industrial-strength theorem prover specialised for the Z notation. Apart from our having in-house expertise, these are two provers of different families (i.e. LCF [Pau94] and Boyer-Moore [KMM09]), which we think will highlight the issues and differences between proof styles and strategies.

We are ensuring models denote, hence undefinedness will not participate in proofs to follow. A study on whether proofs from different logics transfer across theorem provers is an interesting subject in itself and has been discussed in [WF08] regarding using the Z/EVES prover to discharge VDM proof obligations.

1.1 Inferring proof intent

We collect meta-information about models and proof scripts throughout the proof development process. Some of this information, such as expected typing features or expected signatures (LHS) of lemmas that would discharge or weaken current goals, can be automatically inferred.

Within formal methods (and across different methods like VDM, Z or B), proof obligations tend to have a predictable shape. This repetition in the phrasing of theorems suggests the

[3]See `http://www.overturetool.org`

possibility of repeated proofs. This is corroborated in practice, providing proof experts are available. Our aim is to de-skill this process by, given a set of proofs from an expert and the data we collect, provide proof support to discharge the remainder (similar/familiar) proof obligations.

User annotations might declare specific (and open-ended) proof intent, such as existential-witnessing often appearing in feasibility proofs, domain-element mapping for well-definedness in proofs involving maps, type-definition morphisms, etc., are also part of this process. We want to capture the "*Why*s" within various decisions taken by the user. For example, was a particular representation of a data type used for convenience, previous experience, ease of proof, or something else?; what kind of extra annotation to add to the description of the problem that would help improving proof automation?; how to inspect the proof traces of different provers to infer/measure the quality of (different) formulations?; *etc.* Given that most time spent on proof involves failure of some kind, we are more interested in the theorem proving processes leading to the final proof, rather than just the final (usually polished/optimised) proof scripts and theory representations.

The aim is to detect the most relevant meta-proof information needed to characterise, and eventually infer, proof strategies and/or suggest lemmas. That is, to infer possible lemmas to suggest, and indeed proof script snippets to reproduce/adjust given (structurally repeated) scenarios on different problems. We call this our language of how to say "*Why*" within a proof step/scenario (see Chapter 3). These abstract reasons on why certain steps were taken are then used to prune the possible proof search space.

We are interested in capturing the modifications in the model, as well as the "*aha*" moments within a proof (ie., those—often final—steps leading to a neat solution). Within research reports from AI4FM, we have an initial catalog of such "*whys*" to be used for pruning the proof search space, in particular with reasons/ideas coming from rippling [BBHI05a] and from a set of proof scenarios such as: identification of induction within goals; proof chunking (or problem splitting); n-proofs (*e.g.* *n*—different—proofs from same goal); cut-rules (*e.g.* lemma identification and introduction); goal generalisation and anti-unification; etc. Thus, failed proof attempts are as important as the final proof script: they contain the thought process towards the end result. Hopefully, given our previous experience with proof of large scale models [FW08, BFW09], and enough proof data collected by both extra proof annotations and by listening to the interactions between users and theorem provers, we will be able learn proof strategies of interest.

We are also investigating the use of machine learning to mine useful features from this data in a process akin to what is described in preliminary tools in this area[4]. The meta-information collected is guided by a formal development, again using VDM and proof, where more details are in [JFV13]. Our tools extend the Eclipse platform by embedding theorem provers of interest in the background, such that we are able to eavesdrop on the interactions between the user writing specifications, failing at proof obligations and then the changing the shapes of lemmas or models, within the theorem proving system used in the process.

1.2 Proof engineering

We call the streamlining of such proof processes *proof engineering*. That is, before we can tackle any proof obligation born from modelling, we first need to shape and polish models to fit the needs of a mechanical theorem prover, yet at the same time, keep faithful to the original design intent (*i.e.* no model adjustment for the sake of an easier proof). We claim that this setup is crucial for the successful mechanisation of any industrial-scale specification regardless of specific method or prover. In Chapter 4, we discuss how we systematically performed such steps for the heap model within the context of both Z/EVES (see Appendix G) and Isabelle/HOL (see

[4]See `http://www.computing.dundee.ac.uk/staff/katya/ML4PG/`

Chapter 5); it explores two specification methods and two theorem provers, which are quite different in nature: Z is described with untyped classical logic, whereas VDM used the Logic of Partial Functions; Z/EVES belongs within the Boyer-Moore family of theorem provers, whereas Isabelle/HOL belongs to the LCF (Logic of Computable Functions) family.

We see the exercise with these variations as crucial, since it illustrates the generality of our ideas. Even though proof strategies and lemma suggestions for Z might not transfer across provers as readily as across formalisms — this is not that surprising. We have empirical evidence that these techniques are transferable to other notations, like VDM or B, and other theorem provers.

Importance of lemmas Before one can get to the nub of the problem within industrial-scale proof obligations, which almost always involve large formulae (i.e. tens of pages long) and multiple (i.e. over 100) variables, we claim it is fundamental to have in place a considerable amount of machinery to enable automation to an acceptable level. Proof engineering is essential for scalability: it takes a good amount of unrelated proof effort in order enable one to tackle the actual proof obligations of interest. Lemmas are useful whenever one needs to either: decompose a complex problem; fine-tune the theorem prover's rewriting abilities to given goals; generalise a solution of some related (usually more abstract) problem; and to provide alternative solutions/encodings of the same data structure/algorithm being modelled; *etc*[5].

1.3 Proof obligations in formal methods

Well-formedness proofs involve application of partial functions and uniqueness of existential quantifiers. Their complexity is directly proportionate to the complexity of involved data types. In the abstract specification of the heap, unnecessarily difficult auxiliary functions are used (see Chapter 2). Overcoming the difficulty these data representation choices make for proof is a key part of the proof process, although what it achieves is not immediately visible: the appropriate setup of lemmas and type morphisms is not perceptible until the top-level POs are discharged. If only one inspected the proof traces leading up to the successful (neatly constructed final) proof script!

Arguably, if proof mechanisation was in mind, the heap might have been modelled differently. That is not the point, though. The point is, given a model "warts and all", what can expert use of tools do to improve proof automation, or indeed point out issues to engineers where automation is likely to fail? Once such process is in place, the well-formedness proofs become relatively straightforward.

Feasibility proofs are harder: they require finding witnesses for the after state and outputs providing the before state invariants, the inputs and the operation predicates themselves. Beyond just proof engineering over types, extra lemmas exposing key relationships between the state and operation invariants are often necessary. And this can make the proof effort more difficult. Refinement proofs establish a link between abstract and concrete specifications and tend to be repetitive and tedious. They are also the most complex of the POs of interest discussed here.

1.4 Outline

The rest of this report is organised as follows:

- Chapter 2 explains our (final) model of two levels of representation of the Heap storage problem. It describe the issues regarding the data representation invariant, as well as

[5]This list is not exhaustive, but those that came up during the development involving Tokeneer's abstract specification [CB08].

how the first refinement from levels 0 to 1 is made. It is directly related to Chapter 4, which represents and evolution of the models from the original in VDM [JS90, Chapter 7], through various versions in both Z/EVES (Appendix G) and Isabelle (Chapter 5).

- Chapter 3 describes the AI$_4$FM contribution to how to represent proof processes by means of bodies of knowledge. They represent meta-level structure and information regarding dependencies between theories and problems of interest. These models are at the heart of our Eclipse proof processing tools (see Section 7.2).
- Chapter 4 describes the evolution between models, from the originals through to the current one in Chapter 2.
- Chapter 5 describes the encoding of the model in Chapter 2 into Isabelle. This includes resolving issues of data type representation, such as data type invariants and VDM maps, within Isabelle's type system. We elided discussion about undefinedness, and we assume the models to be well-formed, except at places like partial functions such as *locs-of*, which we use the Isabelle *undefined* marker, which should never appear in the middle of proofs.
- Chapter 6 discusses the proofs resulting from the efforts of Chapter 5.
- Chapter 7 presents our conclusions, points for discussion and future work.
- Appendix G presents the links to various resources related to the Z/EVES models and proofs discussed partially in Chapter 4.

Chapter 2

Modelling heap storage

In this chapter we present a VDM development of a HEAP memory manager. From an initial abstract specification, a design is given at increasing levels of (representation) detail.

The original specification and development from [JS90, § 7] is discussed in Chapter 4, together with a historical perspective on how and why this original model evolved to what is summarised in this Chapter.

We chose the HEAP problem as it is well known, is abstractly simple, and has some non-trivial refinement proofs (albeit with relatively easy retrieve functions between state representations). The original VDM development is given as a textbook example for modelling and refinement using VDM and it does refer to some of the involved refinement proofs (i.e. different state representations are compatible), yet little is said about feasibility (i.e. operation preconditions are strong enough to to ensure that the postconditions are satisfiable) or well-formedness (i.e. functions are applied within their domains) proofs, or any sanity checks (i.e. desirable properties of the model). In our development, we discharge proof obligations related to these four levels of consistency checking across different layers of data refinement.

We remained as faithful as reasonable to George's original model, up to the point where design decisions appeared to us to be questionable or mistaken. What we did not do was to change the model just to make proofs easier; we believe that in larger industrial models that is not a practical option and that difficulties in proof are better tackled by introducing lemmas etc.

2.1 Heap as a set of (contiguous) locations (level 0)

The specification (Level 0) of a HEAP store manager offers two simple operations $NEW0$ requests an allocation of s contiguous bytes and $DISPOSE0$ frees a specific contiguous sequence of bytes. The state of this abstract specification is simply a set of locations:

$$Free0 = Loc\text{-}\mathbf{set}$$

The issue of handling adjacency is handled by accepting that Loc is synonomous with $\mathbb{N}$.

The basic function for constructing a range of memory locations given an initial location and size is called *locs-of*. At this level, memory is modelled as a set of locations:

$$locs\text{-}of : Loc \times \mathbb{N}_1 \to Loc\text{-}\mathbf{set}$$

$$locs\text{-}of(l, n) \triangleq \{l, \ldots, l + n - 1\}$$

A predicate *is-block* is also defined to verify if a memory range exists in a set of locations:

$$is\text{-}block : Loc \times \mathbb{N}_1 \times Loc\text{-}\mathbf{set} \to \mathbb{B}$$

$$is\text{-}block(l, n, ls) \triangleq locs\text{-}of(l, n) \subseteq ls$$

The heap operations at level 0 are defined next. They use *locs-of* and *is-block* to construct the necessary range of memory locations and update the state accordingly. For *NEW*, we return a single location—the starting location of the memory allocated—across all levels with the assumption that allocated location sizes will be respected. The state is updated by removing the set of allocated locations from the free store. For *DISPOSE*, we ensure that the locations being returned are not already free and perform the inverse operation (union) to add the deallocated memory back to the free store.

$$
\begin{array}{l}
NEW0\ (s{:}\,\mathbb{N}_1)\ r{:}\,Loc \\
\textbf{ext wr}\ f_0\ :\ Free0 \\
\textbf{pre}\ \exists l \in Loc \cdot is\text{-}block(l, s, f_0) \\
\textbf{post}\ is\text{-}block(l, s, \overleftarrow{f_0}) \wedge \\
\qquad f_0 = \overleftarrow{f_0} - locs\text{-}of(r, s) \\
\\
DISPOSE0\ (l{:}\,Loc, s{:}\,\mathbb{N}_1) \\
\textbf{ext wr}\ f_0\ :\ Free0 \\
\textbf{pre}\ locs\text{-}of(l, s) \cap f_0 = \{\,\} \\
\textbf{post}\ f_0 = \overleftarrow{f_0} \cup locs\text{-}of(l, s)
\end{array}
$$

Comments on the model and proofs. We note that zero-memory request are not possible due to the type constraints on the input s. Furthermore, we note that, whilst the precondition for *DISPOSE* is not actually required to satisfy the postcondition at this level, we have it at this level to document the design decision that one cannot deallocate memory that hasn't been allocated.

Finally, the feasibility proofs for both operations are trivial at this level (cf. Chapter 6). For *NEW*, the existential on the precondition provides an appropriate witness for the result, and the updated state is defined in terms of the postcondition, providing a trivial witness. For *DISPOSE*, we simply need to instantiate the updated state as described by the postcondition.

2.2 Heap as a disjoint map of location sizes (level 1)

Level 1 reifies the representation of the heap store by representing the free store as a mapping from locations to their corresponding sizes. This naturally filters out duplicate locations of different sizes, simplifies the non-abuttness/ordering property description and introduces the appropriate level of development regarding allocation ordering. The new state invariant requires that every mapped location is "disjoint" and "separate" (i.e. locations are ordered and non-abutting).

$$
\begin{array}{l}
is\text{-}disj : X\text{-}\mathbf{set} \times X\text{-}\mathbf{set} \to \mathbb{B} \\
is\text{-}disj(s, t) \ \triangleq\ s \cap t = \{\,\} \\
\\
Free1 = Loc \xrightarrow{m} \mathbb{N}_1 \\
\textbf{inv}\ (f) \triangleq \\
\quad (\forall l, l' \in \mathbf{dom}\, f \cdot \\
\qquad l \neq l' \ \Rightarrow\ is\text{-}disj(locs\text{-}of(l, f(l)), locs\text{-}of(l', f(l')))) \wedge \\
\quad \forall l \in \mathbf{dom}\, f \cdot (l + f(l)) \notin \mathbf{dom}\, f
\end{array}
$$

The definition of $NEW1$ in terms of this mapping is given explicitly over the after state depending on whether the requested size is exact or within what is available ($f_1(l) \geq s$), where mapping operations are used to perform appropriate update as domain filtering ($\lhd$) or map union ($\cup$), which is only defined for maps with disjoint domains. The use of map union makes proofs harder because of the domain condition that the maps are disjoint. Using map override ($\dagger$) would lead to much easier proofs and would not make much difference to the model given

the precondition of both operations already guarantee map domain disjointness. Insisting on using union where the domains are known to be disjoint makes a fact about the model clear.

$NEW1\ (s{:}\,\mathbb{N}_1)\ r{:}\,Loc$
ext wr f_1 : $Free1$
pre $\exists l \in \mathbf{dom}\, f_1 \cdot f_1(l) \geq s$
post $r \in \mathbf{dom}\ \overleftarrow{f_1}\ \wedge$
$(\overleftarrow{f_1}(r) = s \wedge f_1 = \{r\} \lhd \overleftarrow{f_1}\ \vee$
$\overleftarrow{f_1}(r) > s \wedge f_1 = (\{r\} \lhd \overleftarrow{f_1}) \cup \{r + s \mapsto \overleftarrow{f_1}(r) - s\})$

The complexity of ordering non-abutting locations is brought to the surface in the definition of both $NEW1$ and $DISPOSE1$. For the latter, we make the design decision of finding adjacent locations to be merged that might be either above and below the location being returned, which may be empty. Adjacent location mappings are merged as ***ext***ended set calculated by their minimum location to be mapped to the sum of all mapping sizes involved, including the ones being returned. The auxiliary functions calculating minimum location and summed sizes are defined recursively on the cardinality of the domain of the map, which is finite in VDM.

$DISPOSE1\ (d{:}\,Loc, s{:}\,\mathbb{N}_1)$
ext wr f : $Free1$
pre $is\text{-}disj(locs\text{-}of(d,s), locs(f))$
post $\exists below, above, ext \in Loc \xrightarrow{m} \mathbb{N}_1 \cdot$
$below = \{l \mid l \in \mathbf{dom}\, f \wedge l + \overleftarrow{f}(l) = d\} \lhd f \wedge$
$above = \{l \mid l \in \mathbf{dom}\, f \wedge l = d + s\} \lhd f \wedge$
$ext = above \cup below \cup \{d \mapsto s\} \wedge$
$f = (\mathbf{dom}\, below \cup \mathbf{dom}\, above \lhd \overleftarrow{f}) \cup$
$\{min\text{-}loc(ext) \mapsto sum\text{-}size(ext)\}$

where:

$min\text{-}loc : (Loc \xrightarrow{m} \mathbb{N}_1) \to Loc$
$min\text{-}loc(sm) \triangleq$ **if** $sm = \{x \mapsto y\}$
then x
else let $x \in \mathbf{dom}\, sm$ **in** $min(x, min\text{-}loc(\{x\} \lhd sm))$
pre $sm \neq \{\,\}$

$min : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$min(x,y) \triangleq$ **if** $x < y$
then x
else y

$sum\text{-}size : (Loc \xrightarrow{m} \mathbb{N}_1) \to \mathbb{N}$
$sum\text{-}size(sm) \triangleq$
if $sm = \{\,\}$
then 0
else let $x \in \mathbf{dom}\, sm$ **in** $sm(x) + sum\text{-}size(\{x\} \lhd sm)$

Comments on model and proof On the strictly greater case for $NEW1$, we make a design decision to choose the right-most section of contiguous memory to be the one allocated (*i.e.* the maplet update as $\{r + s \mapsto \overleftarrow{f_0}(r)\text{-}s\}$). We could have also defined a (more abstract) non-deterministic choice among any of the possible contiguous set of locations, which would lead $NEW1$ to be the exact inverse of $DISPOSE1$. We chose a specific implementation without

realising this observation. In retrospect, this is likely to be simplifying the proofs involving $NEW1$, yet our decision was oblivious to this fact[1].

The explicit commitment to the design decision of finding adjacent locations to return makes the proof strategies about this model clearer. In $DISPOSE1$, the extended map can have at most three elements with respect to the returned amount (s) for given location (d). This makes the proof strategy for discharging the overall goal modular, where the complexity of chasing further adjacent pieces is naturally separated by the invariant, which helps identifying strategies to reuse from other proofs involving like those from [FW09].

Although this model seems more complicated, its proof obligations are still relatively straightforward (cf. Chapter 6). For instance, the feasibility witnesses are almost trivial, given the one-point rule applies for all variables involved on both operations, if the precondition is split at the right (= and >) cases for $NEW1$.

This is mostly to do with explicit aspects of the invariant being separate and directly defined, rather than implicitly described. The use of case distinction over the precondition of $NEW1$ as either equal or strictly smaller requested sizes help discharging the goal, but also clearly declare the intent behind what is being modelled.

2.3 Feasibility

At each level we prove well-formedness and feasibility of operations using two theorem provers (with the model represented in their notations). Proofs at level 0 are straightforward, given there is no state invariant. There are no well-definedness proofs as the auxiliary functions are total; and there are (standard) feasibility proof obligations per operation, where pre/post conditions are expanded in place from definitions (see Appendix A). The PO for the feasibility of $NEW0$ is:

$$\boxed{NEW0\text{-feas}}\ \frac{\overleftarrow{f_0} \in Free0,\ s \in \mathbb{N}_1 \qquad pre\text{-}NEW0(\overleftarrow{f_0}, s)}{\exists f \in Free0, r \in Loc \cdot post\text{-}NEW0(s, \overleftarrow{f_0}, f, r)}$$

where the theorem stated in a prover looks like this:

$$\forall s \in \mathbb{N}_1, \overleftarrow{f_0} \in Free0 \cdot pre\text{-}NEW0(s, \overleftarrow{f_0}) \Rightarrow \exists r \in Loc\text{-}\mathbf{set}, f_0 \in Free0 \cdot post\text{-}NEW0(s, \overleftarrow{f_0}, f_0, r)$$

which is equivalent to

$$\forall s \in \mathbb{N}_1, \overleftarrow{f_0} \in Free0 \cdot \exists l \in Loc^* \cdot is\text{-}block(l, s, \overleftarrow{f}_0) \Rightarrow \exists r \in Loc\text{-}\mathbf{set}, f_0 \in Free0 \cdot \exists s \in Loc^* \cdot (is\text{-}block(r, s, \overleftarrow{f_0}) \wedge f_0 = \overleftarrow{f_0} - locs\text{-}of(r, s))$$

For given inputs (s) and before state ($\overleftarrow{f_0}$), outputs (r) and after state (f_0) need to be found ($\exists$), providing the precondition is strong enough ($\Rightarrow$) to establish the postcondition. The PO for the feasibility of $DISPOSE0$ is similar:

$$\boxed{DISPOSE0\text{-feas}}\ \frac{\overleftarrow{f_0} \in Free0,\ d \in Loc,\ s \in \mathbb{N}_1 \qquad pre\text{-}DISPOSE0(d, s, \overleftarrow{f_0})}{\exists f \in Free0 \cdot post\text{-}DISPOSE0(d, s, \overleftarrow{f_0}, f)}$$

[1] This design decision also appears in the widen-precondition proof of the reification between levels 0 and 1 in Section 6.3.2

At level 1, the proofs of "feasibility" for the two operations are mildly challenging, if lengthly, because the invariant on *Free*1 has to be maintained. The PO for the feasibility of *NEW*1 is:

$$\boxed{NEW1\text{-feas}}\ \frac{\overleftarrow{f_1} \in Free1, s \in \mathbb{N}_1 \quad pre\text{-}NEW1(s, \overleftarrow{f_1})}{\exists f_1 \in Free1, r \in Loc \cdot post\text{-}NEW1(s, \overleftarrow{f_1}, f_1, r)}$$

The precondition hypothesis shows that a location exists and might suggest a case split for the inner disjunction in the conclusion but it is likely that a theorem proving (TP) system will need help to spot this. Since f is defined using only total operators it is clearly defined and of the correct (unconstrained) type. So the only difficulty is —as expected— the invariant.

2.4 Sanity checks

Proving feasibility of our operations does not guarantee adherence to the "expected" interaction or behaviour. We have several identities that give us further assurance. The first property is to show that directly disposing space is always possible

$$\boxed{NEW0\text{-}DISPOSE0\text{-Post}}\ \frac{\begin{array}{l}\overleftarrow{f}, f \in Free0, r \in Loc, s \in \mathbb{N}_1 \\ pre\text{-}NEW0(\overleftarrow{f}, s) \\ post\text{-}NEW0(\overleftarrow{f}, s, f, r) \\ pre\text{-}DISPOSE0(f, r, s)\end{array}}{post\text{-}DISPOSE0(f, r, s, \overleftarrow{f})}$$

$$\boxed{NEW0\text{-}DISPOSE0\text{-Pre}}\ \frac{\begin{array}{l}\overleftarrow{f}, f \in Free0, r \in Loc, s \in \mathbb{N}_1 \\ pre\text{-}NEW0(\overleftarrow{f}, s) \\ post\text{-}NEW0(\overleftarrow{f}, s, f, r)\end{array}}{pre\text{-}DISPOSE0(f, r, s)}$$

Another example is that 'new applied twice cannot return the same location'; or, more generally: the locations do not intersect:

$$\boxed{NEW0\text{-}NEW0\text{-disjoint}}\ \frac{\begin{array}{l}f1, f2, f3 \in Free0, r1, r2 \in Loc, s1, s2 \in \mathbb{N}_1 \\ pre\text{-}NEW0(f1, s1) \\ post\text{-}NEW0(f1, s1, f2, r1) \\ pre\text{-}NEW0(f2, s2) \\ post\text{-}NEW0(f2, s2, f3, r2)\end{array}}{is\text{-}disj(locs\text{-}of(r1, s1), locs\text{-}of(r2, s2))}$$

Another property could relate recently freed heap space:

$$\boxed{DISPOSE0\text{-}NEW0\text{-Pre}}\ \frac{\begin{array}{l}\overleftarrow{f}, f \in Free0, l \in Loc, s1, s2 \in \mathbb{N}_1 \\ pre\text{-}DISPOSE0(\overleftarrow{f}, l, s1) \\ post\text{-}DISPOSE0(\overleftarrow{f}, l, s1, f) \\ s1 \geq s2\end{array}}{pre\text{-}NEW0(f, l, s2)}$$

That is, claiming a new amount of space just after freeing a portion of at least that size is always possible. This is just a subset of the properties possible. These properties are proved for both levels (0 and 1) of development (see Chapter 6).

2.5 Reification POs

The reification proofs use the following retrieve function linking the data representation at each level of development (from 0 to 1):

$$retr0 : Free1 \to Free0$$

$$retr0(f) \triangleq locs(f)$$

In VDM, reification induces three different types of proof obligation. First, a single *adequacy* PO states that every level 0 state can be mapped to a level 1 state through the retrieve function:

$$\boxed{Free1\text{-adequacy}}\ \frac{f_0 \in Free0}{\exists f_1 \in Free1 \cdot f_0 = retr0(f_1)}$$

The second type of proof obligation is *widen precondition*, which states that (for *DISPOSE*):

$$\boxed{DISPOSE1\text{-w-pre}}\ \frac{\begin{array}{l} f \in Free1, l \in Loc, s \in \mathbb{N}_1 \\ pre\text{-}DISPOSE0(retr0(f), l, s) \end{array}}{pre\text{-}DISPOSE1(f, l, s)}$$

The third type is *narrow postcondition*, which (for *DISPOSE*) states:

$$\boxed{DISPOSE1\text{-n-post}}\ \frac{\begin{array}{l} \overleftarrow{f}, f \in Free1, l \in Loc, s \in \mathbb{N}_1 \\ pre\text{-}DISPOSE0(retr0(\overleftarrow{f}), l, s) \\ post\text{-}DISPOSE1(\overleftarrow{f}, l, s, f) \end{array}}{post\text{-}DISPOSE0(retr0(\overleftarrow{f}), l, s, retr(f))}$$

Chapter 3

Models of why

This chapter enlarges on [JFV13] in motivating and describing an abstract model of the AI$_4$FM system. Significantly, the extension of the earlier material uses parts of the example in Chapter 2.

The overall architecture of an AI4FM system can be seen in Figures 3.1 and 3.2.

Figure 3.1 pictures how high-level strategies will be "captured" in AI$_4$FM. The numbered arcs are explained as follows:

1. Having a record of why a conjecture is being tackled, the system can attempt to "parse" any interactions initiated by the expert against existing strategies.
2. The expert will be asked to name any new strategies and be invited to mark identifying features.
3. The system can note undischarged goals, record success/failure of strategies; and record the lemmas that are used.
4. The system can suggest strategies to the expert.

Similarly, the extra indexed arcs in Figure 3.2 relate to the "replay" of strategies and are explained:

1. The system can replay (possibly modified versions of) strategies that fit the context and have been previously generated in expert mode. As explained below, an attempt is made to order the use of options based on previous success/failure.
2. Success/failure of strategies is noted both to trigger a move to the next option and to adjust weights that will affect future choices. If necessary, failure of the final option will cause the system to backtrack to an earlier point in the proof tree.
3. The system must keep the user informed (especially about backtracks); it might also ask about lemmas.
4. The engineer might be able to assist if automatic attempts (just) fail; alternatively, there might be a need to bring an expert on line.

To realise this functionality data has to be stored in AI4FM; this chapter presents an abstract (VDM) model of the state which we believe can achieve the information gathering. We are here, following the approach used when *mural* [JJLM91] was developed: we are thinking out the architecture in terms of an abstract model thereof. The model itself is contained in Section 3.5.1; the sections that precede the model try to build up the case for the various components in an intuitive way.

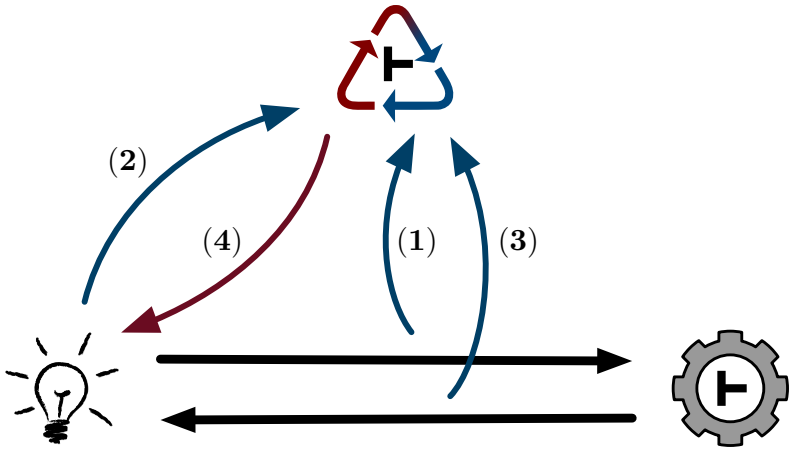

Figure 3.1: AI4FM "capture" mode

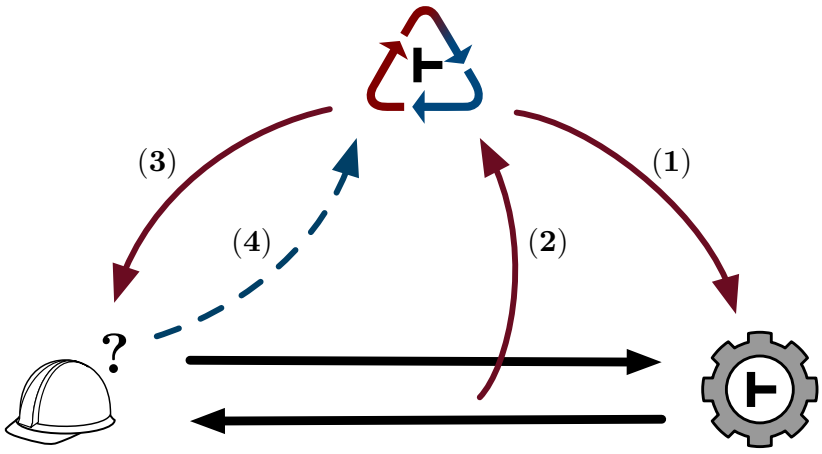

Figure 3.2: AI4FM "replay" mode

The conjectures and proof content of a body (Section 3.2) is fairly routine; Section 3.3 on "strategies" is central to the realistion of the AI$_4$FM hypothesis. Firstly, the overall structure of the data is described in Section 3.1.

3.1 Bodies of knowledge

The accumulated information in an AI$_4$FM instantiation can be thought of as a collection of bodies (in the sense of "body of knowledge").

$$\Sigma :: bdm : BdId \xrightarrow{m} Body$$
$$\cdots$$

There will be bodies of knowledge about mathematical theories such as set theory (cf. Section 3.1.1); there will also be bodies that relate to single specifications (cf. Section 3.1.2). (Relationships between bodies (*bdrels*) are used in finding strategies and are discussed in Section 3.4.)

A *FnDefn* contains the signature of the function and, optionally, its definition in terms of more basic operators. Thus far:

$$\begin{array}{lll} Body :: uses & : & BdId\text{-}\mathbf{set} \\ \quad functions & : & FnId \xrightarrow{m} FnDefn \\ \quad \cdots & & \\ FnDefn :: type & : & Signature \\ \quad defn & : & [Definition] \end{array}$$

3.1.1 Base theories (as *Body* objects)

Consider, say, the *Body* for sets of "locations" as in the model in Section 2.1. The *BdId* will be some memorable name such as LocSet. It will "use" the generic theory for sets and that for *Loc* (the polymorphic theory for X-**set** will in turn use that for $\mathbb{N}$ for the result of **card** s).

For illustration, assume that the body for LocSet introduces the new function (this could as well be in the generic theory of sets.)

$$disj : Loc\text{-}\mathbf{set} \times Loc\text{-}\mathbf{set} \to \mathbb{B}$$
$$disj(s1, s2) \quad \triangleq \quad s1 \cap s2 = \{\}$$

This signature and definition are stored in a *FnDefn*.

Similarly, there would be a body (of knowledge) for $Loc \xrightarrow{m} \mathbb{N}_1$ (cf. Section 2.2).

3.1.2 Specifications give rise to bodies

As well as the general theories in Section 3.1.1, we would also expect each (VDM) user specification to be linked to a *Body* corresponding to its "state". Thus there will be more than one *Body* associated with the HEAP problem (cf. Chapter 2) — at least one per refinement layer and separate ones for reifications that connect refinements.

So one *Body* of interest in Chapter 2 is that for HEAP1. This will record that it *uses* base types such as sets and maps; it will also contain definitions of the predicate *inv-Free*1 and the functions *all-locs* and *locs-of*.[1]

We might go further. The example in Chapter 2 is unusual in that a series of "non-record" states suffice for the development. In examples such as those from the industrial partners in the DEPLOY project, states of 20 fields were not unusual — and these states also had lengthy invariants. Issue 4 indicates that more in-built support for records might be required. In most industrial cases, the state will be defined as a record. A trivial case such as:

[1] For the purposes of this chapter, there has been some refactoring [Whi13] wrt Chapter 2: the function *locs* has been renamed *all-locs* $d1$ has been renamed d etc. This is particularly important for Section 3.2.3.

$$X :: f1 : T1$$
$$\quad\; f2 : T2$$

would give rise to constructor and selector functions:

$mk\text{-}X\colon T1 \times T2 \to X$
$f1\colon X \to T1$
$f2\colon X \to T2$

Beyond that, it might be worth generating sub-theories for any separable sub-states (in the sense that data type invariants and/or operations force some fields to be grouped together — other than these constraints, models should be broken down as far as is possible). The examples in Chapter 2 are unfortunately not large enough to illustrate this.

3.2 Proof objects

This section describes the information that AI4FM has to retain about proofs themselves. This might appear to duplicate what is going on in the ATP but looking again at Figures 3.1 and 3.2 it should be clear that AI4FM has to retain knowledge of any proof tasks that either are still open or which were open and whose completion was achieved with the help of AI4FM; where the ATP can discharge a PO automatically, only that fact need be stored.

The conjectures and proof content of a body is similar to that in the formal description of *mural* (see [JJLM91]); Proof objects are those entities related to proof process analysis which is detailed in the coming sections.

3.2.1 Conjectures

The information in a *Body* that is of use in proofs is the collection of formal results that are built up over the lifetime of that body.

$$Body :: \cdots$$
$$\quad guts : ConjId \xrightarrow{m} Conjecture$$
$$\quad \cdots$$

The *guts* of a body is a collection of proof tasks (*Conjecture*). A proof task has hypotheses and a goal both containing judgements. A *Judgement* can be typing information, a sequent or an equation. In addition there can be any number of (attempts at) justifications. Thus:

$$Conjecture :: hyps : Judgement^*$$
$$\quad goal : Judgement$$
$$\quad status : \{\textsc{Lemma}, \textsc{RewriteL2R}, \textsc{NegativeProperty}, \cdots\}$$
$$\quad justifs : JusId \xrightarrow{m} (\textsc{Axiom} \mid \textsc{Trusted} \mid Justification)$$
$$\quad \cdots$$
$$Judgement = Typing \mid Sequent \mid Equation \mid Ordering \mid \cdots$$

An example of a low level conjecture would be a natural deduction proof rule for "or elimination": it would have hypotheses $E_1 \vdash E$, $E_2 \vdash E$ and $E_1 \vee E_2$ and a conclusion of E. This conjecture might be marked as an axiom (Axiom). (Where there is nothing on the left of a sequent, the convention of dropping the $\vdash$ is followed.) Another might record that if $S_1, S_2, S_3\colon X\text{-}\mathbf{set}$, $S_1 \subseteq S_2$ and $S_2 \subseteq S_3$ then $S_1 \subseteq S_3$. This conjecture might be marked as trusted (Trusted) in the sense that it came from a trusted source document.

Within a body for a specification (cf. Section 3.1.2), a proof obligation generator will create a *Conjecture* for each proof obligation (PO) about the consistency of that single specification. Proof obligations will also be generated corresponding to the claim that one model reifies another (obviously this has to be triggered by the claimed reification link) e.g. Section 6.3.

It is important to remember that the first action for any conjecture is to pass it to at least one ATP: AI4FM has nothing to do if, say, Isabelle discharges the PO. We might also arrange that counter examples are sought if the first attempt at proof fails.

To return to the body for LocSet, the following straightforward lemmas are likely to be *Judgement*s (ultimately accompanied by a justification).

$$\boxed{\text{L1}}\ \frac{\begin{array}{l} s1, s2{:}\ X\text{-}\mathbf{set} \\ \mathit{disj}(s1, s2) \\ s3 \subseteq s2 \end{array}}{\mathit{disj}(s1, s3)}$$

$$\boxed{\text{L1.5}}\ \frac{s1, s2{:}\ X\text{-}\mathbf{set}}{\mathit{disj}(s2, s1 - s2)}$$

$$\boxed{\text{L2}}\ \frac{\begin{array}{l} \mathit{disj}(s1, s3) \\ \mathit{disj}(s2, s3) \end{array}}{\mathit{disj}((s1 \cup s2), s3)}$$

The use of lemmas is a crucial element in conducting proofs at an appropriate "level of discourse" (cf. Leo's term "zooming"). In contrast, it would be technically possible –when proving results about actual models– to just expand out the definition of *disj* but this would obscure proofs of the results about say HEAP1. Even more obfuscating would be to expand, for example, set subtraction via its definition in terms of predicates. Conducting a proof at a high level of discourse if nearly always better than going to a lower level and lemmas are the key way of achieving this.

The main activity of a user is to discharge POs and it is precisely here that strategies become important. But it should be remembered that the first thing that happens to any PO is that it is fed to the (or more than one) "theorem prover of choice": if, for example, Isabelle discharges the PO only that fact is stored. This happens for *NEW*0-feas and *DISPOSE*0-feas from Section 6.2.

The conjectures of a body generated from a specification will contain all of the proof obligations (e.g. invariant preservation, links between models, etc.). Appendix A gives the general form of rules for VDM POs.

One top-level PO from *HEAP*1 would be:

$$\boxed{\mathit{DISPOSE}1\text{-feas}}\ \frac{\begin{array}{l} \overleftarrow{f_1} \in \mathit{Free}1, d \in \mathit{Loc}, s \in \mathbb{N}_1 \\ \mathit{pre\text{-}Dispose}1(d, s, \overleftarrow{f_1}) \end{array}}{\exists f_1 \in \mathit{Free}1 \cdot \mathit{post\text{-}DISPOSE}1(d, s, \overleftarrow{f_1}, f_1)}$$

Section 3.2.3 traces through a justification of a lemma that is needed to discharge this PO and Section 3.3.2.1 explains the strategies involved.

3.2.2 Justifications

Turning to *Justification*, remember that it is explicitly envisaged that there can be multiple attempts to justify a proof task (*i.e.* *Conjecture*s can have a mapping to different *Justification*s). When a conjecture is first generated, it will have no justifications. A user might start one proof justification, leave it aside and try another, then come back and complete the first proof. But notice that the notion of whether a proof is complete (in the sense of (transitively) relying only on axioms) is a complex recursive predicate.

Overall,

$$\begin{array}{lll} \mathit{Justification} :: & \mathit{claim} & : (\mathit{ConjId} \mid \mathit{ToolOP}) \\ & \mathit{subst} & : \mathit{Term} \xrightarrow{m} \mathit{Term} \\ & \mathit{sub\text{-}probs} & : \mathit{ConjId}\text{-}\mathbf{set} \end{array}$$

A justification which uses an established inference rule will point to its *ConjId*. The *subst* relates the terms in the inference rule to those in the *hyps* and *goal* of the *Conjecture*. The *sub-probs* field points to any sub-problems that need to be discharged to complete the proof. Notice that such a justification corresponds to one step in a proof: collecting a whole proof requires tracing the attempts at the sub-conjectures. A low-level instance of *Justification* might record that the rule on which it is based (*rule*) is "and elimination right":

$$\boxed{\wedge\text{-Er}}\ \frac{E_1 \wedge E_2}{E_1}$$

(much more interesting would be the use of an induction rule — but the same structure applies); in this case the hypotheses (*hyps*) will point to a single conjecture that is a conjunction but almost certainly with large expressions as conjuncts; the substitution (*subst*) will relate the E_i to the components of the conjecture pointed to by *hyps*.

In *rare* cases, proof steps can be as fine-grained as in *mural* [JJLM91] — the example that follows is unrealistic in the sense that we'd certainly expect any TP system to handle it automatically. The classic instance of a strategy is case split; $\vee$-E is the obvious example. One nice property of $\vee$-E is that, by pointing at a disjunct, the decomposition is clear. The important point is that the proof of one *Conjecture* can give rise to several others. This in turn shows that we need to define a predicate that can check whether a proof is complete and a function that can help a user locate incomplete proof tasks. Similar comments apply to both $\forall$NEEDED, $\exists$NEEDED. The former then creates a place for the various parts of induction. Another item that might not be too hard is NORMALFORMREDUCTION. In contrast, CUTRULE gives no clue how to generate an intermediate *Judgement* that is the essence of user intuition in top-down proof. This puts a lot of reliance on what could be detected when the "expert" uses a cut rule to split a proof task. For the time being, we're assuming that most useful examples of the cut rule will have to be annotated by the expert.

In practice, TP tools such as Isabelle and Z/EVES are powerful enough that a user will hardly ever interact at the level of the (natural deduction) laws of the logic itself. So, in fact, the most prevalent examples of *Justification* ought come from the underlying theorem prover; as shown in Figure 3.2. Use of a ATP will be recorded as an instance of *ToolOP* — such output will be specific enough to the specific ATP that it is not further specified here. If it is an SMT tool, the *claim* might be no more than the name of the tool. Notice however that Isabelle's *auto* will generate *sub-probs*.

The *Features* set is described in Section 3.3.

3.2.3 Representing a hand proof

In order to explain the basic way in which detailed proof attempts are handled, this section takes a rather low-level lemma that arose in the Isabelle version of the HEAP proof and looks at how it fits into the *guts* of Σ of Section 3.5.1. The important topic of how strategies help in constructing this proof is postponed to Section 3.3.2.1.

A lemma that arises during the proof of feasibility of *DISPOSE*1 concerns a situation where $below = \{\} \wedge above \neq \{\}$[2] — for brevity, this is referred to as "L99":

$$\boxed{\text{L99}}\ \frac{inv\text{-}Free1(f);\ disj(locs\text{-}of(d,s),\ all\text{-}locs(f));\ d+s \in \mathbf{dom}\, f}{disj(locs\text{-}of(d, s+f(d+s)),\ all\text{-}locs(\{d+s\} \lhd f))}$$

[2]The lemma here is one of the four subgoals within *z_F1_inv_dispose1_Disjoint* (l. 757 of HEAP1Proofs.thy). This is part of the *DISPOSE*1 feasibility proof: postcondition update Disjoint invariant subgoal 5? 1. PO-DISPOSE 2. PO-DISPOSE-POST 3. PO-DISPOSE-POST-DISJOINT 4. PO-DISPOSE-POST-DISJOINT-LEMMA-APPL 5. SUBGOAL 5 of that case when $below = \{\} \wedge above \neq \{\}$.

This will be a *Conjecture* (that happens not to get proved automatically by Isabelle) in an appropriate *Body* but first let's look at the *Body* that contains information about (finite) maps from *Loc* to $\mathbb{N}_1$.

We assume that the following basic lemmas have been proved (their numbering is to do with the order in which they arose):

$$\boxed{\text{L3}}\ \frac{\begin{array}{l} m\colon \mathit{Loc} \xrightarrow{m} \mathbb{N}_1 \\ s\colon \mathit{Loc}\text{-}\mathbf{set} \end{array}}{\mathit{all\text{-}locs}(s \lhd m) \subseteq \mathit{all\text{-}locs}(m)}$$

$$\boxed{\text{L3.5}}\ \frac{\begin{array}{l} m\colon \mathit{Loc} \xrightarrow{m} \mathbb{N}_1 \\ s \in \mathbf{dom}\, m \\ \forall l, l' \in \mathbf{dom}\, m \cdot l \neq l' \ \Rightarrow\ \mathit{disj}(\mathit{locs\text{-}of}(l, m(l)), \mathit{locs\text{-}of}(l', m(l'))) \end{array}}{\mathit{all\text{-}locs}(s \lhd m) = \mathit{all\text{-}locs}(m) - \mathit{locs\text{-}of}(s, m(s))}$$

$$\boxed{\text{L4}}\ \frac{\begin{array}{l} d\colon \mathit{Loc} \\ n, m\colon \mathbb{N}_1 \end{array}}{\mathit{locs\text{-}of}(d, n+m) = \mathit{locs\text{-}of}(d, n) \cup \mathit{locs\text{-}of}(d+n, m)}$$

These three lemmas are stored as *Judgement*s. Here again, establishing apposite lemmas[3] ensures that the proofs about the model itself can be conducted at a high level of discourse.

Presented as an outline of a natural deduction proof, the *Conjecture* of interest (mapped to by *L*99) is:

$$\begin{array}{ll} \textbf{from} & \mathit{inv\text{-}Free}1(f); \mathit{disj}(\mathit{locs\text{-}of}(d, s), \mathit{all\text{-}locs}(f)); d + s \in \mathbf{dom}\, f \\ \textbf{infer} & \mathit{disj}(\mathit{locs\text{-}of}(d, s + f(d+s)), \mathit{all\text{-}locs}(\{d+s\} \lhd f)) \qquad ?? \end{array}$$

When created, this will have an empty collection of *justifs* — this is indicated above by the "??" where one would expect to see a justification. As repeatedly stated above, the first thing to do is to let one or more TPAs have a go at proving the conjecture. Unsurprisingly (see specific functions such as *all-locs* and a non-trivial invariant), Isabelle fails to discharge this automatically.

The Isabelle transcript in Appendix C shows that the expert made several attempts before completing the proof; this is precisely why *justifs* is a mapping. For brevity in this first exposition, a "perfect" proof is envisaged (but see Section 3.3.2.1).

Lemma L4 provides a convenient equality to expand $\mathit{locs\text{-}of}(d, s+f(d+s))$ in the hypothesis. This effectively completes the proof of *L*99 in that we can fill in its justification — but it has spawned two new unproven conjectures *S*1 and *S*3 (the numbering of steps *Si* is again indicative of the order of creation); it so happens that *S*1 has a side condition but this is immediately discharged leading to the following state of proof:

[3] The proof of L3.5 would use *inv-Free*1 to get:
$\forall l, l' \in \mathbf{dom}\, f \cdot l \neq l' \ \Rightarrow\ \mathit{disj}(\mathit{locs\text{-}of}(l, f(l)), \mathit{locs\text{-}of}(l', f(l')))$
which specialises to:
$\forall l \in \mathbf{dom}\, f \cdot l \neq d + s \ \Rightarrow\ \mathit{disj}(\mathit{locs\text{-}of}(d+s, f(d+s)), \mathit{locs\text{-}of}(l, f(l)))$

from	$\mathit{inv\text{-}Free1}(f); \mathit{disj}(\mathit{locs\text{-}of}(d,s), \mathit{all\text{-}locs}(f)); d+s \in \mathbf{dom}\, f$	
S2	$f(d+s) \in \mathbb{N}_1$	h[4], *Free*1
S1	$\mathit{locs\text{-}of}(d, s+f(d+s)) =$ $\mathit{locs\text{-}of}(d,s) \cup \mathit{locs\text{-}of}(d+s, f(d+s))$	L4(S2)
S3	$\mathit{disj}((\mathit{locs\text{-}of}(d,s) \cup \mathit{locs\text{-}of}(d+s,f(d+s))),$ $\mathit{all\text{-}locs}(\{d+s\} \lhd f))$	??
infer	$\mathit{disj}(\mathit{locs\text{-}of}(d, s+f(d+s)), \mathit{all\text{-}locs}(\{d+s\} \lhd f))$	$= \mathit{subs}(S1, S3)$

Effectively, the current state σ_2 has four conjectures in *guts* — its domain is $\{L99, S3, S1, S2\}$.

Turning to the unjustified *Conjecture* indexed by $S3$; L2 provides a way of splitting this task into two sub-tasks. Thus we can discharge S3 by generating two sub-conjectures S4 and S6.

from	$\mathit{inv\text{-}Free1}(f); \mathit{disj}(\mathit{locs\text{-}of}(d,s), \mathit{all\text{-}locs}(f)); d+s \in \mathbf{dom}\, f$	
S4	$\mathit{disj}(\mathit{locs\text{-}of}(d,s), \mathit{all\text{-}locs}(\{d+s\} \lhd f))$	??
S6	$\mathit{disj}(\mathit{locs\text{-}of}(d+s, f(d+s)), \mathit{all\text{-}locs}(\{d+s\} \lhd f))$	??
S2	$f(d+s) \in \mathbb{N}_1$	h[4], *Free*1
S1	$\mathit{locs\text{-}of}(d, s+f(d+s)) =$ $\mathit{locs\text{-}of}(d,s) \cup \mathit{locs\text{-}of}(d+s, f(d+s))$	L4(S2)
S3	$\mathit{disj}((\mathit{locs\text{-}of}(d,s) \cup \mathit{locs\text{-}of}(d+s,f(d+s))),$ $\mathit{all\text{-}locs}(\{d+s\} \lhd f))$	$L2(S4, S6)$
infer	$\mathit{disj}(\mathit{locs\text{-}of}(d, s+f(d+s)), \mathit{all\text{-}locs}(\{d+s\} \lhd f))$	$= \mathit{subs}(S1, S3)$

Leo confirms my hope that proofs of S4 and S6 are found by Isabelle thus completing the justification. For reference, a natural deduction proof that is a "picture" of the final proof is:

from	$\mathit{inv\text{-}Free1}(f); \mathit{disj}(\mathit{locs\text{-}of}(d,s), \mathit{all\text{-}locs}(f)); d+s \in \mathbf{dom}\, f$	
S5	$\mathit{all\text{-}locs}(\{d+s\} \lhd f)) \subseteq \mathit{all\text{-}locs}(f)$	L3, *Free*1
S4	$\mathit{disj}(\mathit{locs\text{-}of}(d,s), \mathit{all\text{-}locs}(\{d+s\} \lhd f))$	L1(S5, h[2])
S7	$\mathit{all\text{-}locs}(\{d+s\} \lhd f) =$ $\mathit{all\text{-}locs}(f) - \mathit{locs\text{-}of}(d+s, f(d+s))$	L3.5, h[1], h[4]
S6	$\mathit{disj}(\mathit{locs\text{-}of}(d+s, f(d+s)), \mathit{all\text{-}locs}(\{d+s\} \lhd f))$	L1.5, S7
S2	$f(d+s) \in \mathbb{N}_1$	h[4], *Free*1
S1	$\mathit{locs\text{-}of}(d, s+f(d+s)) =$ $\mathit{locs\text{-}of}(d,s) \cup \mathit{locs\text{-}of}(d+s, f(d+s))$	L4(S2)
S3	$\mathit{disj}((\mathit{locs\text{-}of}(d,s) \cup \mathit{locs\text{-}of}(d+s,f(d+s))),$ $\mathit{all\text{-}locs}(\{d+s\} \lhd f))$	$L2(S4, S6)$
infer	$\mathit{disj}(\mathit{locs\text{-}of}(d, s+f(d+s)), \mathit{all\text{-}locs}(\{d+s\} \lhd f))$	$= \mathit{subs}(S1, S3)$

The numbering of proof steps above is, of course, different from their linear order; the numbering indicates something of the creation order but this should be clear from the text of the preceding section.[4]

Two Isabelle versions of this proof are contained in Appendix C.

[4] It is of course possible to build a larger strategy by combining several steps.

3.3 Strategies

3.3.1 Data

The capture, modification and replay of strategies is central to AI4FM (cf. Chapter 1 and Figures 3.1/3.2). Such strategies are part of a *Body*:

$$\begin{array}{lll} \textit{Body} :: \cdots \\ \qquad & \textit{strats} & : \textit{StrId} \xrightarrow{m} \textit{Strategy} \end{array}$$

The purpose of a strategy is to progress proofs. This can be done either by using a tool or by using a previously extracted strategy. (Tools can either be part of the ATP of choice or can by separately developed "apps" within AI$_4$FM— e.g. [GKL13].)

$$\begin{array}{lll} \textit{Strategy} :: \textit{function} & : & (\textit{ToolIP} \mid \cdots) \\ \qquad \cdots \end{array}$$

As with *ToolOP* above, the input required by different tools will vary and it is difficult to pin down its content beyond:

$$\begin{array}{lll} \textit{ToolIP} :: \textit{name} & : & \cdots \\ \qquad \textit{support} & : & \textit{ConjId}\text{-}\mathbf{set} \\ \qquad \textit{other} & : & \cdots \end{array}$$

We have made a special case of identifying that some tools –such as SMT solvers– will require a selection of lemmas as *support*.

Previously acquired strategies will split a problem

$$\begin{array}{lll} \textit{Strategy} :: \textit{function} & : & (\cdots \mid \textit{Split}) \\ \qquad \textit{justif} & : & \textit{ConjId} \\ \qquad \cdots \end{array}$$

$$\textit{Split} = \textit{Conjecture} \to \textit{Conjecture}\text{-}\mathbf{set}$$

Just as in all LCF-like systems, the flip side of *split* is the *justif* that proves the decomposition.

Notice that *split* is a (general) function (cf. Section 3.3.4). Were it the case that –each time a new strategy was conceived– there was a programmer "in the loop", a new chunk of code would realise the split of a proof task (*Conjecture*) into sub-tasks. But in AI4FM we want to achieve the learning process without a programmer in the loop. So one possibility is that a single, more general, chunk of code could analyse previous uses of a strategy and figure out the required split. This code will essentially be trying to generalise (to the stored function) the instance that the "expert" has just executed. In an example like "multi-base-case" induction, this generalisation should not be too difficult to spot but this clearly requires more thought in general.

A low level strategy might involve splitting a problem into sub-cases; another could reduce an expression to a normal form; an important collection of strategies will be for induction; an interesting form might shift the representation of an object of interest to a different body of knowledge.

The identification of the most useful strategy builds on the (repeated) "why" of our writings (e.g. [JFV13]).

$$\begin{array}{lll} \textit{Strategy} :: \cdots \\ \qquad \textit{intent} & : & [\textit{Why}] \\ \qquad \textit{rank} & : & \textit{Conjecture} \to \textit{Score} \\ \qquad \cdots \end{array}$$

The set *Why* will never be closed — a user can always add a new concept — examples of *Why* are listed in Section 3.5.2.1. Notice that there is a layering among the strategies — see below on their taxonomy.

The collection of strategies can be thought of as representing "and" and "or" information. The "or" function is represented by having alternative strategies. For example, we do not explicitly say that StructuralIndn, $\mathbb{N}$PeanoIndn and $\mathbb{N}$CompleteIndn are options —

they are just three strategies that might be applicable in similar circumstances. The choice ("or") function is, in a sense, underneath the covers for the user (it might be pursued by (limited) parallelism).

An "and" split in a *Strategy* shows that in order to justify a conjecture, multiple sub-conjectures must be discharged (although in some cases it will just be a reformulation and generate only one sub-task — e.g. contrapositives of implications, use of an isomorphic model — of course, at the leaves of a strategy there are no sub-tasks).

Strategies can be organised into a "taxonomy". The idea is perhaps best illustrated by an example:

$\mathbb{N}$PeanoIndn specialises InductionProof
$\mathbb{N}$CompleteIndn specialises InductionProof

So the final field becomes:

Strategy :: $\cdots$
specialises : [*StrId*]

3.3.2 Selecting strategies

This section builds the bridge from the data structure in Section 3.3.1 to the arcs numbered 1 and 3 in Figure 3.2.

Remember that "in the beginning", there will be no strategies and few lemmas! Clearly, there has to be some "seeding" by an initial set of strategies to make an AI$_4$FM system useful.

Of course, the whole point of recording strategies is to be able to replay them to provide justifications for new conjectures. So the situation considered here is that automatic proof has failed. At the top level, a conjecture will be a PO and the names of the POs are contained in *Why* — so AI4FM can look for strategies that have been captured (see Section 3.3.4). Typically, a strategy just decomposes a proof task to several (hopefully simpler) conjectures. Here again, the first step is to see if the chosen TP(s) can discharge these. If not, strategies for the sub-problems are sought.

We are assuming that the most specific strategy is the most promising: where it fails, an option is to go to the next less "specific" strategy (cf. *specialises*).

The order in which strategies are tried is governed by its *Score* and this is an area where we hope to use some form of "machine learning". If/when all options for a strategy have proved fruitless, AI4FM will be able to trace a higher point in the "proof tree" and try alternatives from there.

When the TP system fails, the process of exploring known strategies starts. As indicated, the *Origin* of the PO is an important guide.

Given a collection of strategies, we need a way of selecting the one that is most likely to succeed. This is a place where machine learning ought be of use.

The applicability of a particular strategy to a putative result (*Conjecture*) is to be evaluated by the *test* function

Strategy :: $\cdots$
rank : *Conjecture* $\rightarrow$ *Score*
$\cdots$

Notice that *rank* is a (general) function; it will be a piece of code that uses the weights learnt in application.

The way in which we envisage learning playing a part in the deployment of strategies is in the function from *Conjecture* to *Score* (which latter is just some ordered set). This will choose how to weight information contained in the *Features* part of *Conjecture*.

Conjecture :: $\cdots$
match : *Features*

The data on which such learning will be based will be something like:

$$\begin{array}{lll} \mathit{Features} :: \mathit{provenance} & : & (\mathit{Origin} \mid \mathit{Why})^* \\ \mathit{mainTps} & : & \mathit{BdId}\text{-}\mathbf{set} \\ \mathit{mainFns} & : & \mathit{FnId}\text{-}\mathbf{set} \\ \mathit{blocks} & : & \mathit{ConjId}\text{-}\mathbf{set} \\ \mathit{other} & : & \cdots \end{array}$$

A conjecture that arises directly from POG will have a *provenance* which contains the name of the POG as it's *provenance* (for any development method, there will be at least one strategy for each class of PO). As steps are made the *Why* entries of strategies that have been applied are concatenated to the provenance fields of the subsidiary conjectures.

Furthermore, in the above:

- knowing what gave rise to a particular conjecture is expected to be key for matching; if a *Why* is contained here, it indicates that the conjecture came from a named strategy
- the types of *mainTps* and *mainFns* indicate what they contain but the pragmatics are more interesting — we think the user often knows that "the action" is on something deeply embedded in a formula — for now, we're assuming that the user will mark these manually
- we have discussed trying to analyse where a conjecture gets blocked — i.e. which generated sub-conjectures get blocked
- within *Features*, the *other* area might include things like the number of operators — hopefully, many/most of these can be extracted automatically. Leo and Cliff have joked about *other* containing information like the number of coffees the user has drunk that session — the point is that new factors can arise that were not planned when the AI4FM instantiation was initiated.

3.3.2.1 Strategies and the example in §3.2.3

Looking back at the proof in Section 3.2.3, we can now consider how strategies might match the evolving proof task. At the first step, there is a general strategy to "rephrase" a conjecture. The user can use this strategy by choosing an equality that (hopefully) simplifies a goal by substitution. There are two non-trivial expressions in the goal that could be expanded, the user tries $\mathit{locs\text{-}of}(d, s + f(d + s))$ perhaps because there is a convenient lemma (L4). This effectively completes the proof of the overall goal but spawns two new unproven conjectures S1 and S3. It so happens that S1 has a side condition but this is automatically discharged leading to the state described as σ_2 in Section 3.2.3.

Isabelle still fails to find a proof. Here there is a generic "split" strategy. Of course, the simplest instance of this generic strategy is "and introduction"; but L2 has the same shape[5] and what the "split strategy" needs is a way of decomposing a conjecture into some number (here two) simpler conjectures. Thus we can discharge S3 by generating two sub-conjectures S4 and S6.

Leo confirms my hope that S5 and S7 in the full proof are found by Isabelle thus completing the justification.

As mentioned above, two Isabelle versions of this proof are contained in Appendix C.

3.3.2.2 Outline of a higher-level example

A strategy that is used in several of the proofs about HEAP (examples listed below to explain how they match the general strategy) is to "uncover a hidden case distinction" (this is its

[5] Don't be fooled by the shape of the symbol in the conclusion of L2 it does work like an "and introduction" rule.

intent). It is useful when a proof has to proceed by cases but there is no obvious disjunction in the *hyps* of the current *Conjecture*. The clue is often hidden in the *goal* of the *Conjecture*.

Take, for example, *NEW*1-*feas*. Here *post-NEW*1 has a disjunction on whether $\overleftarrow{f_1}(r)$ is greater than or equal to the requested size s. What is needed here is a special case of the *cut* strategy which generates the disjunction that is not visible in the hypotheses of the current conjecture. Once this is found the generic strategy for "reasoning by cases" can take over.

In the case of *NEW*1-*feas*, the new hypothesis is neither hard to discover nor prove (it follows from the *pre-NEW*1).

The proof of *DISPOSE*1-*feas* benefits from the same strategy but *post-DISPOSE*1 hides the case distinction more thoroughly. Here, the essential distinction is whether *below* and/or *above* are empty or singleton maps.

The same strategy is again invaluable to the validity proof that *NEW*1 followed by *DISPOSE*1 is an identity over *Free*1.

3.3.2.3 Example: HEAP feasibility POs

In Section 2.3, we discuss the feasibility proof obligations for the heap model. How might the meta-modelling in this chapter help? Well, the above *Conjecture* has a *provenance* of [VPO-Feas, Expand]. We might have a *Strategy* for case distinctions that looked at relations like $\geq$ and proposed a split into $>, =$. If this is the first time we've used it in this context, the expert might have to fire it; but if so, the knowledge will be added that this can be a useful strategy in this context. (If this has been done before, presumably it will get a good *Score* and get selected automatically.) Either way, the strategy ought spawn two conjectures — the first being:

$$\frac{\begin{array}{l}\overleftarrow{f} \in (Loc \xrightarrow{m} \mathbb{N}_1), s \in \mathbb{N}_1 \\ \forall l, l' \in \mathbf{dom}\ \overleftarrow{f} \cdot sep(l, l', \overleftarrow{f}) \\ \exists l \in \mathbf{dom}\ \overleftarrow{f} \cdot \overleftarrow{f}(l) = s\end{array}}{\begin{array}{l}\exists f \in (Loc \xrightarrow{m} \mathbb{N}_1), r \in Loc \cdot \\ \quad r \in \mathbf{dom}\ \overleftarrow{f} \wedge \\ \quad \overleftarrow{f}(r) = s \wedge f = \{r\} \lhd \overleftarrow{f} \wedge \\ \quad \forall l, l' \in \mathbf{dom}\, f \cdot sep(l, l', f)\end{array}}$$

Now, we want to assume this still doesn't go through by some automatic tool. (this might be pessimistic but our assumption lets us make a point about lemmas). Providing the TP system still fails, we assume the expert offers a lemma:

$$\frac{\begin{array}{l}f_1 \in (Loc \xrightarrow{m} \mathbb{N}_1), s \in \mathbb{N}_1 \\ r \in \mathbf{dom}\, f_1 \\ f_1(r) = s \\ \forall l, l' \in \mathbf{dom}\, f_1 \cdot sep(l, l', f_1) \\ f_2 = \{r\} \lhd f_1\end{array}}{\forall l, l' \in \mathbf{dom}\, f_2 \cdot sep(l, l', f_2)}$$

assuming *sep* is defined as the second predicate in the *Free*1-*inv* definition.

The second spawned conjecture from the case split would be:

$$\frac{\begin{array}{l}\overleftarrow{f} \in (Loc \xrightarrow{m} \mathbb{N}_1), s \in \mathbb{N}_1 \\ \forall l, l' \in \mathbf{dom}\ \overleftarrow{f} \cdot sep(l, l', \overleftarrow{f}) \\ \exists l \in \mathbf{dom}\ \overleftarrow{f} \cdot \overleftarrow{f}(l) > s\end{array}}{\begin{array}{l}\exists f \in (Loc \xrightarrow{m} \mathbb{N}_1), r \in Loc \cdot \\ \quad r \in \mathbf{dom}\ \overleftarrow{f} \wedge \\ \quad \overleftarrow{f}(r) > s \wedge f = \overleftarrow{f} \dagger \{r \mapsto \overleftarrow{f}(r) - s\} \wedge \\ \quad \forall l, l' \in \mathbf{dom}\, f \cdot sep(l, l', f)\end{array}}$$

Now, assuming this gets stuck in the same way, AI4FM ought be able to notice that a lemmas is a "good thing"; is it wildly optimistic to expect that we can detect the earlier pattern and spot that the "right" lemma should be:

$$\frac{\begin{array}{l}f_1 \in (Loc \xrightarrow{m} \mathbb{N}_1), s \in \mathbb{N}_1 \\ r \in \mathbf{dom}\, f_1 \\ f_1(r) > s \\ \forall l, l' \in \mathbf{dom}\, f_1 \cdot sep(l, l', f_1) \\ f_2 = f_1 \dagger \{r \mapsto f_1(r) - s\}\end{array}}{\forall l, l' \in \mathbf{dom}\, f_2 \cdot sep(l, l', f_2)}$$

This lemma involves arithmetic and we are less sure what TP systems will make of it. The PO for *DISPOSE*1 is similar.

3.3.3 Facing lemma gaps

The proof discussed in Sections 3.2.3 and 3.3.2.1 was simplified by the presence of useful lemmas. A key issue for a no-expert user is the difficulty of predicting what lemmas will be useful. Without the lemmas, a user can well stumble into nested proofs that –for example– bring in confusing extra quantifiers.

There is much more experimentation required here. One hope is that replaying old strategies will be able to prompt where they rely on lemmas.

Our basic position is that we ought to be able to do something with a good (specific) strategy or an apposite lemma — but can't achieve much if both are missing. Of course, if we have both, we'd hope that the proof would go through.

3.3.4 Capturing strategies

This section expands on Figure 3.1.

After a failure to obtain an automatic proof, we "call an expert" (or even: have a cup of coffee, go for a walk, etc.). Assuming we chose the right expert, she says "it's obvious" and makes a different choice in some step of the proof.

This new choice has to be captured. Generalising from a specific split performed by the expert might not be as difficult as we/I feared. The expert is faced with a recalcitrant conjecture; she performs a specific split; this means that we have the goal and several hypotheses at hand; finding a generalisation of these (that takes the specific instance) to a more general *split* function for the new *Strategy* is something like "anti-unification" (but cbj is pretty sure it is not identical). Of course, said expert might be able to provide an even more useful generalisation. My guess is that the *justif* field will often be filled in as TRUSTED at first — with a proof of the new strategy being provided later (maybe when reviewed?).

Hopefully, the expert comes up with a better name than EUREKA but actually more important is providing the *specialises* link. I don't yet see any way of generating this automatically.

A different, but related, scenario is where the expert stares at the troublesome conjecture and decides that a lemma is the key to progress. Clearly, AI4FM needs to capture the fact that the process was moved forward by finding/generating a lemma. Again, we need experiments

but my hunch is that useful generalisations will be harder here than with simple splits. For example, an "equivalent" lemma might involve operators in a different theory. In fact, I think this is back to the territory of *bdrels*.

3.3.5 An analogy (formally known as "Leo's 2c worth")

Perhaps a useful analogy to interpret *Conjecture* is to discuss the notion of (and difference from) a tactic. In a prover like Isabelle, a tactic is an (ML) program taking a list of goals (as Isabelle *Terms*), which include hypothesis, together with a justification function (as an ML program named as part of a proof script) [AD10]. Such tactic returns a new set of goals with an updated justification, until it reaches **true** and *thm*. In MWhy, goals (and hypothesis) are *Judgment*s representing the term language, with extra (meta-level and structural) information added, whereas *Justification* are brought as a series of (proof) attempts through proof scripts (*Attempt*) or external tools (*Tool*), *etc.* (see next Section). This way, we are bringing to the surface of user modelling intent information about the way tactics might change or update goals.

For instance, if *Justification* brings to the surface the user intent of a proof attempt, the *status* of a conjecture is given by the user as (structural) meta-level information for the prover about the way the user expects the conjecture to be used. This is already present in various provers as tags associated with declarations. In Isabelle, the user can give to definitions and lemmas various kinds of status, such as simplification, introduction, elimination, congruence, transitivity rule and so on. Similarly, in Z/EVES the user can tell the prover whether the lemma is to be used as a proof context (hypothesis) enhancer, hence influence forward proof steps, or else as a backward chaining (goal matching) rewrite rule.

3.4 Relations between bodies

The final part of the state:

$$\begin{aligned} \Sigma :: \mathit{bdm} &: \mathit{BdId} \xrightarrow{m} \mathit{Body} \\ \mathit{bdrels} &: \mathit{BdId} \xrightarrow{m} (\mathit{BdId} \times \mathit{Relationship})\text{-}\mathbf{set} \end{aligned}$$

concerns relationships between bodies of knowledge. Like *Why* itself, this will have to be expandable by the user. Some examples that we can see include:

$$\mathit{Relationship} = \mathit{Specialisation} \mid \mathit{Morphism} \mid \mathit{Isomorphism} \mid \mathit{Inherits} \mid \mathit{Sub} \mid \mathit{Similarity} \mid \cdots$$

We might, for example, have some very abstract items in *Body* such as Larch's "collector"; sets, sequences and maps would all then be specialisations of collector. Another abstract item might be "inductable" — it is here that the more general knowledge about setting up inductive proofs would reside.

Morphism and *Isomorphism* will be used for precise mathematical relationships — the latter for where results can be used in either direction.

Similarity will be for less precise connections (fuzzy matches).

In all of these cases, the idea is that inspiration for a proof strategy might come from a related body of knowledge.

3.5 Summary of "Model"

3.5.1 Data structure

$$
\begin{array}{l}
\Sigma :: \mathit{bdm} : \mathit{BdId} \xrightarrow{m} \mathit{Body} \\
\qquad \mathit{bdrels} : \mathit{BdId} \xrightarrow{m} (\mathit{BdId} \times \mathit{Relationship})\text{-}\mathbf{set} \\
\mathit{Body} :: \mathit{uses} : \mathit{BdId}\text{-}\mathbf{set} \\
\qquad \mathit{domain} : \{\textsc{Rail}, \textsc{Auto}, \ldots\} \\
\qquad \mathit{functions} : \mathit{FnId} \xrightarrow{m} \mathit{FnDefn} \\
\qquad \mathit{guts} : \mathit{ConjId} \xrightarrow{m} \mathit{Conjecture} \\
\qquad \mathit{strats} : \mathit{StrId} \xrightarrow{m} \mathit{Strategy} \\
\mathit{FnDefn} :: \mathit{type} : \mathit{Signature} \\
\qquad \mathit{defn} : [\mathit{Definition}] \\
\mathit{Conjecture} :: \mathit{hyps} : \mathit{Judgement}^* \\
\qquad \mathit{goal} : \mathit{Judgement} \\
\qquad \mathit{status} : \{\textsc{Lemma}, \textsc{RewriteL2R}, \textsc{NegativeProperty}, \cdots\} \\
\qquad \mathit{justifs} : \mathit{JusId} \xrightarrow{m} (\textsc{Axiom} \mid \textsc{Trusted} \mid \mathit{Justification}) \\
\qquad \mathit{match} : \mathit{Features} \\
\mathit{Judgement} = \mathit{Typing} \mid \mathit{Sequent} \mid \mathit{Equation} \mid \mathit{Ordering} \mid \cdots \\
\mathit{Justification} :: \mathit{claim} : (\mathit{ConjId} \mid \mathit{ToolOP}) \\
\qquad \mathit{subst} : \mathit{Term} \xrightarrow{m} \mathit{Term} \\
\qquad \mathit{sub\text{-}probs} : \mathit{ConjId}\text{-}\mathbf{set} \\
\mathit{ToolOP} = \cdots \\
\mathit{Features} :: \mathit{provenance} : (\mathit{Origin} \mid \mathit{Why})^* \\
\qquad \mathit{mainTps} : \mathit{BdId}\text{-}\mathbf{set} \\
\qquad \mathit{mainFns} : \mathit{FnId}\text{-}\mathbf{set} \\
\qquad \mathit{blocks} : \mathit{ConjId}\text{-}\mathbf{set} \\
\qquad \mathit{other} : \cdots \\
\mathit{Origin} = \mathit{Token} \\
\mathit{Why} = \mathit{Token} \\
\mathit{Strategy} :: \mathit{function} : (\mathit{ToolIP} \mid \mathit{Split}) \\
\qquad \mathit{justif} : \mathit{ConjId} \\
\qquad \mathit{intent} : [\mathit{Why}] \\
\qquad \mathit{rank} : \mathit{Conjecture} \to \mathit{Score} \\
\qquad \mathit{specialises} : [\mathit{StrId}] \\
\mathit{ToolIP} :: \mathit{name} : \cdots \\
\qquad \mathit{support} : \mathit{ConjId}\text{-}\mathbf{set} \\
\qquad \mathit{other} : \cdots \\
\mathit{Split} = \mathit{Conjecture} \to \mathit{Conjecture}\text{-}\mathbf{set} \\
\mathit{Why} = \mathit{Token} \\
\mathit{Relationship} = \mathit{Specialisation} \mid \mathit{Morphism} \mid \mathit{Isomorphism} \mid \\
\qquad\qquad \mathit{Inherits} \mid \mathit{Sub} \mid \mathit{Similarity} \mid \cdots
\end{array}
$$

3.5.2 Discussion of the model

3.5.2.1 Comments on some specific elements of *Origin*/ *Why*

The origins of conjecture are important in selecting an appropriate strategy. The set *Origin* will include names of POGs for a method (e.g. VPO-ADEQUACY, VPO-WIDENPRE, VPO-RESTRICTPOST, ZPO-COMPUTEPRE).

Indications of what a strategy is "good for" (*Why*) will include:

EXTRACTSUBSTATE**:** This is here as reminder that large records can confuse a ATP system (if only because of the number of selector/constructor functions and lemmas relating them) — so a useful strategy is to split a large (state) record into independent (wrt the invariant) chunks and to have properties for using results on the sub-states to draw conclusions about the whole state.

$\forall$NEEDED is an example of a *Why* with an obvious strategy but it might be worth taking the step via SETUPINDUCTION and INDUCTIONPROOF; remember also that an alternative strategy could be to apply de Morgan's law.

$\exists$NEEDED is similar — and GENWITNESS is one potential sub-strategy (but so is applying de Morgan's laws).

SETUPINDUCTION**,** INDUCTIONPROOF**,** INDUCTIONRULE are general — more specific are $\mathbb{N}$PEANOINDN and $\mathbb{N}$COMPLETEINDN — these are there to remind us that there is more than one way to do induction over the natural numbers.

DISTRIBUTEOPERATORS**,** COMMUTEOPERANDS**, etc.** should be obvious and might be sub-strategies of NORMALFORMREDUCTION

CASESPLIT**,** $\vee$**-E** are just reminders of low-level strategies.

CUTRULE could be problematic — see Section 3.1.1.

3.5.2.2 Minor clarifications

1. After a discussion with Aaron Sloman, we were considered storing important non-fact such as that list concatenation is, in general, not commutative. We hoped that this would provide clues as to when, say, properties from set theory should not be sought in the theory of sequences. We're now minded to store
   $\exists s, t \in \mathbb{N}^* \cdot s \frown t \neq t \frown s$

   but to mark its *status* as NEGATIVEPROPERTY.

2. One issue for the implementation is that the large recursive function for checking whether a proof is "complete" (in the sense that all subsidiary conjectures are axioms or TRUSTED) could be made more efficient by some form of "memoising".

3. It will be useful to be able to locate *instances* of strategy use — but, for the time being at least, the model stores the pointers in the other direction (see *Features*).

4. We found in *mural* [JJLM91] that records (in the VDM sense) can be difficult in that there is really a different *Body* for each record shape. But records are so ubiquitous that we have to do something for them and we do not favour expanding out "axioms" for all of the constructors/selectors.

5. Earlier internal notes have suggested that inference rules can usefully be generated from (recursive) function definitions (as done in various LPF papers [JLS12]); these would also be examples of *Tool* justifications.

6. As indicated in Footnote 4, it is possible to build "multi-step" strategies. Remember that:

$$\frac{A \vdash B;\ B \vdash C}{A \vdash C}$$

is likely to hold for any logic we want to use. So it is possible to build larger strategies from multiple steps. There might well be a problem with the number of combinations: we don't want to store all possible (matching) pairs.

7. With any ATP systems that can be persuaded to disgorge its (incomplete?) proofs, we would have extra material from which AI4FM could learn.

8. There is a question about how much we are prepared to use/control parallel attempts: in Section 3.3.2, the discussion is simplified by assuming a sequential deployment of strategies — obviously this is a choice where many-core (and/or clouds) could prompt reconsideration.

3.5.2.3 Known issues in the model

The Σ model in Section 3.5.1 should be regarded as "work in progress"[6] — some of the issues that we are still debating include:

1. Probably the most surprising aspect of our current model –at least to anyone schooled in "tactic thinking"– is that *Attempt* deals with a single step in a proof. Before going into more detail, it is worth re-reading Point 6 of Section 3.5.2.2: inference steps can be made as high level as the user wishes.

   There is, however, still a case for expressing strategies that consist of sequential composition, case splits and repetition in for example the style of [GKL13]. The place in our model for such expressions is in *ToolInv*.

   Our suspicion is that any such expression of procedural startegies will be more "brittle" than strategies that are matched to the current situation.

   On the other hand, it must be conceded that developing a useful hierarchy of strategies as envisaged in the current Σ will require great taste and care.

   This still leaves the question of how scripts in either approach are learnt — but only experimentation will show which form is easier to learn.

2. We have discussed at various stages the idea that some putative lemmas could be spotted by looking at the form of recursive function definitions.

3. We've deliberately avoided ordering sub-goals in the *split* field of *Strategy*. We're assuming that only graph shape matters but accept that there are cases where order might be important. In fact, echoing J, Leo suggested on an earlier version that any *Conjecture* should be time stamped (one can always write a function that drops this information where not needed). We like this idea but have not yet added it.

4. Thierry Lecomte argued for considering the domain of application in *Features*.

[6] Furthermore, this model is a slight evolution of that in [JFV13].

Chapter 4

How we got to where we are

In this Chapter we describe the process around evolving the model, from the originals presented in Section 4.1 through its modifications done in the Z/EVES model up to the versions within the Isabelle development. These modifications were mostly a combination of error correction and clearer abstractions. For instance, the notions of separability, non-abutingness and disjointness of *Pieces* in original level 1 are all mingled within a single definition of the invariant. Similarly, the retrieve function between level 0 and 1, as well as postconditions for level 1 are suspiciously interlinked for the aid of proof (*i.e.* no design decision is documented in *DISPOSE*1 postcondition, but rather the "right" after state through the retrieve usage).

There were many versions of our (re-)formulation. This was due to both our increasing understanding of the problem and the variations across different theorem provers. The one showed in Chapter 2, and subsequently in Chapter 5, is our final/current version. We kept this history for the sake of exposure of how a typical formal development evolves.

The discussion style described in this chapter is inspired by Naur's description of his solution of Writh's N-Queens problem as described in [Nau72]. Another interesting view / discussion about these models was also developed as an advanced MSc at Newcastle as an pedagogical exercise on the viability of our ideas for a (non-proof expert) well trained engineer [Sle13].

4.1 Models of a heap: VDM originals

The original VDM development of a Heap describes two key operations: *NEW* and *DISPOSE* to allocate and deallocate memory, respectively. The complete development can be found in [JS90, Chapter 7]. It shows how refinement works in VDM by gradually making successive commitments to data structures and algorithms. We chose this as an example given it is a problem of which most programmers are aware whilst still not having trivial proofs.

Firstly, we typeset the models using the VDM Overture tools and fixed a few typos and type errors, like the one in the definition of the *is_sequential* auxiliary function below. Then, using results from [WF08] we encoded the model and proof obligations using the Z/EVES theorem prover to discharge VDM proof obligations within it (see Appendix G).

4.1.1 Heap as a set of locations (L0)

Initially, the heap (level 0) is modelled as abstractly as possible: free-space as a set of locations modelled as natural numbers represents the system state ($Free0$) with no invariant, and the two operations are defined over sets of locations. Recall that the set constructor in VDM represents a finite set.

$$Loc = \mathbb{N}$$

$$Free0 = Loc\text{-}\mathbf{set}$$

The following auxiliary functions are required in the definitions of the operations: they model predicate testing whether a contiguous ($is\_sequential$) sequence of locations ($s \in Loc^*$) of a particular size (n) is within the free memory ($free$)[1].

$$has\_seq : Loc^* \times \mathbb{N} \times Loc\text{-}\mathbf{set} \to \mathbb{B}$$

$$\begin{array}{l} has\_seq(s, n, free) \quad \triangleq \\ \quad \mathbf{elems}\ s \subseteq free \\ \quad \wedge\ \mathbf{len}\ s = n \wedge is\_sequential(s) \end{array}$$

$$is\_sequential : \mathbb{N}^* \to \mathbb{B}$$

$$is\_sequential(s) \quad \triangleq \quad \exists i, j \in \mathbb{N} \cdot \mathbf{elems}\ s = \{i, \ldots, j\}$$

For our purposes, we prove the state has been initialised with a given amount of free memory (i.e. $f_0 = Loc$). Allocation is defined by the next operation ($NEW0$). It is straightforward: providing there is as a contiguous sequence of locations of sufficient length.

$$\begin{array}{l} NEW0\ (req{:}\,\mathbb{N})\ res{:}\ Loc\text{-}\mathbf{set} \\ \mathbf{ext}\ \mathbf{wr}\ f_0\ :\ Loc\text{-}\mathbf{set} \\ \mathbf{pre}\ \exists s \in Loc^* \cdot has\_seq(s, req, f_0) \\ \mathbf{post}\ \exists s \in Loc^* \cdot (has\_seq(s, req, \overleftarrow{f_0}) \wedge \\ \qquad res = \mathbf{elems}\ s \wedge f_0 = \overleftarrow{f_0} - res) \end{array}$$

$$\begin{array}{l} DISPOSE0\ (ret{:}\ Loc\text{-}\mathbf{set}) \\ \mathbf{ext}\ \mathbf{wr}\ f_0\ :\ Loc\text{-}\mathbf{set} \\ \mathbf{pre}\ ret \cap f_0 = \{\,\} \\ \mathbf{post}\ f_0 = \overleftarrow{f_0} \cup ret \end{array}$$

[1] In VDM, sequences are indexed from 1; **len** returns the sequence size (or length); and **elems** returns the sequence (range) values as a set.

To dispose memory, a set of locations is given to be returned to the free state (i.e. it will be updated, **wr** f_0), providing it is not already free (see *pre*). It is assumed that dispose operations will be called only with locations returned by $NEW0$. It takes some (positive) quantity and returns a set of usable memory locations, providing values returned were free, are exactly the size requested and are in sequence (Loc^*). This is modelled using auxiliary function *has_seq*, which finds a sequence of locations of a given size.

Comments on the model. Subsequent refinement proofs performed using Z/EVES highlighted issues with the model. The input to $NEW0$ allows for zero-memory allocation, which does not seem right (*e.g.* why not $\mathbb{N}_1$ instead of $\mathbb{N}$?). The issue surfaces as an extra case split because of overlapping location ranges. The interface of the operations also changes across model levels, which leads to unnecessary complications in the refinement setup. For instance, the result from $NEW1$ is a memory *Piece*, instead of a memory *Loc*ation.

The auxiliary function (*has_seq*) creates a protracted "jump" between types (e.g. from a set to a sequence of locations). In our final reformulation of level 0 (see Section 2.1), we simplify that decision with a clearer (equivalent) notion of free location numeric ranges, which is not only simpler, but also usually has more automation lemmas available.

Comments on proofs. With explicit equations for the after state, proof obligations at level 0 were straightforward in Z/EVES.

4.1.2 Heap as a set of pieces (L1)

Level 1 tackles $NEW0$ inefficiency for a search of a suitable set of locations. The state ($Free1$) is defined as a set of *Piece*s that neither overlap nor abut their corresponding locations, where *Piece* is a two-field record containing a location (of type *Loc*) and size (of type $\mathbb{N}$). In VDM, projection functions are defined for record types (e.g. $LOC(p)$ returns a *Loc* given a $p \in Piece$).

$$
\begin{array}{l}
Piece :: \ LOC \ : \ Loc \\
\qquad\quad SIZE \ : \ \mathbb{N}
\end{array}
$$

$$Free1 = Piece\text{-}\mathbf{set}$$

$$
\begin{array}{ll}
\mathbf{inv}\ (ps) \triangleq & \forall p1, p2 \in ps \cdot \\
& (p1 = p2 \lor locs\_of(p1) \cap locs\_of(p2) = \{\,\} \\
& \land\ LOC(p1) + SIZE(p1) \neq LOC(p2))
\end{array}
$$

The abutting property over $Free1$ is needed in order to ensure that the precondition of $NEW0$ is enough to establish the applicability of $NEW1$ during the refinement proof. The definition of $NEW1$ needs to find a new suitable *Piece*, whereas $DISPOSE1$ returns the locations of a piece to the free set.

$$
\begin{array}{l}
NEW1\ (req\colon \mathbb{N})\ res\colon Piece \\
\mathbf{ext\ wr}\ f_1\ :\ Free1 \\
\mathbf{pre}\ \exists p \in f_1 \cdot SIZE(p) \geq req \\
\mathbf{post}\ locs(f_1) = locs(\overleftarrow{f_1}) - locs\_of(res) \\
\qquad \land\ locs\_of(res) \subseteq locs(\overleftarrow{f_1}) \\
\qquad \land\ SIZE(res) = req
\end{array}
$$

$$
\begin{array}{l}
DISPOSE1\ (ret\colon Piece) \\
\mathbf{ext\ wr}\ f_1\ :\ Free1 \\
\mathbf{pre}\ locs\_of(ret) \cap locs(f_1) = \{\,\} \\
\mathbf{post}\ locs(f_1) = locs(\overleftarrow{f_1}) \cup locs\_of(ret)
\end{array}
$$

This model employs two auxiliary functions that project a set of locations out of *Free*1 and *Piece*. Moreover, *locs* is used as the refinement retrieve function linking *Free*0 (set of locations) to *Free*1 (set of pieces).

$$locs : Free1 \rightarrow Loc\text{-}\mathbf{set}$$
$$locs(ps) \triangleq \bigcup\{locs\_of(p) \mid p \in ps\}$$

Comments on the model. The non-zero sizes and heterogeneous operation interfaces are issues that justify adjusting the model accordingly (i.e. $SIZE \in \mathbb{N}_1$). Beyond the unhelpful non-linear equations, which introduce confusing case-analysis, the non-abutting property of the state invariant introduces the complication of *Piece* ordering. It fails to deal with the case where different pieces share the same location with different sizes, which should not be allowed.

Perhaps the most serious issue is that the operations are defined in terms of the chosen retrieve function (*locs*). This hides the design decision of ordering pieces and introduces complicated existential witnesses over the feasibility of the after state in the proof process for the sake of an easier refinement proof (*i.e.* find an f_1 such that properties of $locs(f_1)$ hold is non-trivial).

Actual design decisions about ordering on the original models are only taken at the last layer of refinement (level 4), which has no proof in [JS90] and is rather complicated. This lead us to reformulate the model with such decisions being explicitly given, instead of modelling to cater for refinement proofs (see final version in Chapters 2, and 5, and intermediate versions in this Chapter and in Appendix G).

Comments on proofs. The proof of feasibility and later refinement between *Free*0 and *Free*1 revealed problems with the *Free*1 invariant as it confuses design decisions within the same defining predicate using non-linear equations, which leads to unnecessarily complicated reasoning during proof. Because the retrieve function (*locs*) between levels is used for specification of level 1, the inadequacy of the invariant does not become immediately apparent. This is due to implicit non-linear equations from the invariant of *Free*1. Given *locs_of* is defined in terms of a range of locations, we prove a weakening lemma[2] stating that from the definition of subrange and *locs_of*, goals involving the *Free*1 invariant can be rewritten as

$$SIZE(p1) = 0 \lor SIZE(p2) = 0 \lor$$
$$LOC(p1) + SIZE(p1) \leq LOC(p2) \lor LOC(p2) + SIZE(p2) \leq LOC(p1)$$

Considering the predicates involved, say in *NEW*1, the before/after state and pre/post conditions lead to 9 non-linear equations with various buried case distinctions to deal with. This lead us to rethink both the invariant, as well as strategies to simplify the proof process. The result was a series of lemmas about algebraic properties of auxiliary functions (see Chapter 3.

4.2 Theorem proving experiences

In the following subsections, we describe the interesting details involved in the way the Z and Isabelle developments of the VDM Heap models evolved. We focus on specific aspects related to model changes or errors, rather than full details. For Z/EVES models, full details can be found in Appendix G. The discussion style here is inspired by [Nau72].

4.2.1 Z/EVES v0 — warts and all

Our first model in Z (see Appendix G) was exactly the same as the original VDM, where auxiliary $\mathbb{B}$-valued functions were defined using set comprehension as usual in Z. They looked

[2] In Isabelle's parlance, this is known as a congruence lemma; and Z/EVES rewrite rule.

like

$$\textbf{is\_ssequential} \mathrel{\widehat{=}} \{s: \mathbb{N}^* \mid \exists i, j \in Loc \cdot \textbf{ran}\,(s) = i \ldots j\}$$

$$\textbf{has\_seq} \mathrel{\widehat{=}} \{s: Loc^*; n: \mathbb{N}; f: Free0 \mid \textbf{is\_sequential}(s) \land \textbf{ran}\,(s) \subseteq f \land \textbf{dom}\,(s) = 1 \ldots n\}$$

Satisfiability proofs at level 0 are trivial. For level one, it leads to the following goal on

$$\ldots \;\Rightarrow\; \exists f \in Free1 \cdot \ldots \land locs(f) = \overleftarrow{locs(f)} - locs\text{-}of(res!) \ldots$$

Finding such witness for f is unnecessarily complicated by the interference of *locs*. And in any case, why should the definition of $NEW1$ be in terms of *locs* instead of manipulating the set of *Piece* that is $Free1$? A similar scenario happens for $DISPOSE1$. When coming to the refinement proof, we also realised about the interface differences between level 0 (talking about size $\in \mathbb{N}$) and level 1 (talking about *Piece*).

4.2.2 Z/EVES v1 — interface and postcondition change

In our first adjustment to the model, we made the interfaces homogeneous by having a *Piece* as output to $NEW0$ and input to $DISPOSE0$. This dispenses the use of auxiliary functions at level 0 and the use of *locs* at level 1. This resolved the issue of interface refinement between levels from the original, and also simplified the notion of location ranges, given that *locs-of* was now defined in terms of *Piece* as

$$locs\_of : Piece \to Loc\text{-}\textbf{set}$$
$$locs\_of(p) \;\;\widehat{=}\;\; \{LOC(p), \ldots, LOC(p) + SIZE(p){-}1\}$$

Added automation lemma on numeric ranges

To aid automation, we also proved a (rewrite rule) lemma saying that

lemma locsOfPiece:
$\forall p \in Piece \cdot locs\text{-}of(p) = LOC(p) \ldots LOC(p) + SIZE(p)$-1

Interface and postcondition modifications to level 0

These modifications make the operations for level 0 look like this:

$NEW0\ (req: \mathbb{N})\ res: Piece$
ext wr f_0 : Loc-**set**
pre $\exists r \in Piece \cdot req = SIZE(r)$
post $SIZE(res) = req \land f_0 = \overleftarrow{f_0} - locs\text{-}of(res)$

$DISPOSE0\ (ret: Piece)$
ext wr f_0 : Loc-**set**
pre $locs\text{-}of(ret) \cap f_0 = \{\}$
post $f_0 = \overleftarrow{f_0} \cup locs\text{-}of(ret)$

The interface to $NEW0$ now returns a *Piece* instead of a Loc-**set**, and its precondition is simpler: no use of auxiliary functions, and instead it depends on finding a suitable size *Piece* (which is the one being returned), instead of a set of locations in the original. The postcondition is similar, but relies on the version of *locs-of* for *Piece*. For $DISPOSE0$, the input is now a *Piece* instead of Loc-**set**. The pre/postconditions are adjusted to use *locs-of* for making a *Piece* into a contiguous set of locations.

Identifying hidden case split in $NEW1$ precondition

For level 1, we the invariant is encoded equivalently to VDM using sets and the operations need modification to avoid having *locs* in the postcondition. At first we wanted to keep the preconditions similar and saw that the equal case for $NEW0$ above. This together with satisfiability proof for $NEW1$ led us to spot the hidden case split on the precondition for $NEW1$.

$$
\begin{array}{l}
NEW1\ (req{:}\,\mathbb{N})\ res{:}\,Piece \\
\textbf{ext wr}\ f_1\ :\ Free1 \\
\textbf{pre}\ \exists r \in f_1 \cdot SIZE(r) \geq req \\
\textbf{post}\ \exists p \in Piece \cdot p \in \overleftarrow{f_1}\ \wedge \\
\qquad locs\text{-}of(res) \subseteq locs\text{-}of(p) \wedge SIZE(res) = req \\
\qquad f_1 = (\overleftarrow{f_1} - \{p\}) \cup (locs\text{-}of(p) - locs\text{-}of(res))
\end{array}
$$

The postcondition is clearly different: instead of relying on *locs* it explicitly removes the new $Piece\ p \in \overleftarrow{f_1}$ to allocated, where the resulting $res \in Piece$ takes just enough out of the piece (p) chosen. Given $locs\text{-}of(res) \subseteq locs\text{-}of(p)$, it is **true** that $SIZE(res) = req$. The result f_1 removes the whole chosen piece p first then adds the remainder amount from p not used by res (*i.e.* when $SIZE(p) > SIZE(res) = req$). This makes explicit the design decision to have the state at level 1 using set of pieces instead of locations. We have also declared two operations with the hidden case split as $NEW1Equal$ and $NEW1Bigger$ and proved that their disjunction is equivalent to $NEW1$. This simplifies the satisfiability proof for $NEW1$ considerably.

Proving satisfiability and lemma discovery

Now the satisfiability witness is trivial through the one point rule. The hard part of this proof for $NEW1$ is to show that the invariant holds for the updated model. The equal case is trivial: $p = res$, hence

$$\ldots \wedge p \in \overleftarrow{f_1}\ \Rightarrow\ (\overleftarrow{f_1} - \{p\}) \cup (locs\text{-}of(p) - locs\text{-}of(res))$$

simplifies to

$$(\overleftarrow{f_1} - \{p\})$$

which is trivially **true**, providing p is instantiated with $r \in \overleftarrow{f_1}$ from the precondition. Nevertheless, this part of the proof led to the following simplification rule lemmas being suggested:

$$
\begin{array}{l}
\textbf{lemma}\ \text{lFree1UnitDiff} \\
\forall f \in Free1, p \in Piece \cdot f - \{p\} \in Free1
\end{array}
$$

$$
\begin{array}{l}
\textbf{lemma?}\ \underline{lFree1UnitUnion} \\
\forall f \in Free1, p \in Piece \cdot \{p\} \cup f \in Free1
\end{array}
$$

The first one states that removing a *Piece* from the state does not violate the invariant, which we prove without difficulty. It is useful in simplifying the $NEW1Equal$ case. The second lemma is not true (in general), and it states you can always add a *Piece* to the state satisfying the invariant. Although this second lemma is not **true**, it brought to our attention the key issue behind the $NEW1Bigger$ proof: **what are the conditions to make singleton extension to $Free1$?**

4.2.2.1 General properties about *locs-of* and *locs*

When we turned to $DISPOSE1$, it became clear that it would be trickier, given the non-abuttingness property would lead to chasing locations potentially to be freed to avoid fragmented memory. In the original, this detail is elided by the cheeky use of *locs*! This led to

the modification on *DISPOSE*1. Nevertheless, in the process we also found some other useful lemmas linking the precondition of *DISPOSE*1 and the *Free*1 invariant, as well as some general properties about *locs-of* and *locs*.

lemma lLocsWithin
$\forall p, q \in Piece \cdot$
$\quad LOC(p) \leq LOC(q) \land LOC(q) + SIZE(q) \leq LOC(p) + SIZE(p)$
$\quad \Rightarrow \ locs\text{-}of(q) \subseteq locs\text{-}of(p)$

lemma lLocsReminder
$\forall rem, res, p \in Piece \cdot$
$\quad LOC(rem) = LOC(p) + SIZE(res) \land$
$\quad SIZE(rem) = SIZE(p)\text{-}SIZE(res) \land$
$\quad SIZE(p) \geq SIZE(res) \land$
$\quad LOC(res) = LOC(p) \ \Rightarrow \ locs\text{-}of(rem) = locs\text{-}of(p) - locs\text{-}of(res)$

Lemmas lLocsWithin and lLocsReminder weakens any goal term involving *locs-of* subset and set difference as a conjunction of non-linear equations. This is useful when dealing with the postconditions of *NEW*1 and *DISPOSE*1. It is also useful as it suggest we might need to think about more general lemmas about *loc-of* and other involved set and map operators, if we are to avoid having to go down to various non-linear equations.

From the lemmas about set union and difference for *Free*1 comes the suggestion for having a similar structure for *locs*, given that it operates on a set of *Piece* just like *Free*1 at this point. So the next two lemmas enforce the *Free*1 invariant through *locs* regarding the two set function symbols involved ($\_ - \_$ and $\_ \cup \_$).

lemma lLocsUnitDiff
$\forall f \in Free1, p \in Piece \cdot$
$\quad p \in f \ \Rightarrow \ locs(f - \{p\}) = locs(f) - locs\text{-}of(p)$

lemma lLocsUnitUnion
$\forall f \in Free1, p \in Piece \cdot$
$\quad locs\text{-}of(p) \cap locs(f) = \{\,\} \ \Rightarrow \ locs(\{p\} \cup f) = locs(f) \cup locs\text{-}of(p)$

4.2.2.2 Lemma shaping and prover technicalities

A technical **note on lemma shapes**: notice that we had the goal conclusion declared quite prescriptively with respect to singleton sets. For instance, we used $\{p\} \cup f \in Free1$ in lFree1UnitUnion instead of $f \cup \{p\} \in Free1$, and we used $locs(\{p\} \cup f) = \ldots$ in lLocsUnitUnion instead of $locs(f \cup \{p\}) = \ldots$. This is deliberate and crucial. And that is because we want to use these lemmas as automatic rewrite rules. Provers tend to rewrite terms like $f \cup \{p\}$ as $\{p\} \cup f$ at the earliest opportunity (*i.e.* first simplification step), hence for our rules to be automatically picked during proof search, they also need to match what the prover expect, despite being unassumingly simple choices such as this.

Conversely, if one wants to "tame" the use of a lemma by the automatic reasoners available, you could explicit state it in a way that would never (automatically) pick up, unless the user fiddles with the goal slightly. For instance, Z/EVES has the lemmas on sequence sizes given as $\mathbf{card}\, s = 0 \ \Leftrightarrow \ s = [\,]$ because one does not want to automatically rewrite every occurrence of $s = [\,]$ into $\mathbf{card}\, s = 0$, yet this is an important useful result.

The same considerations are **true** in Isabelle, if with slight variations in style and level of detail and control.

Rethinking *Free*1 invariant

From the lemmas about *locs-of* above and from the new postconditions for *NEW*1 and *DISPOSE*1, satisfiability proofs entailed a lengthy (and messy) amount of non-linear equations coming from both side conditions of applying weakening lemmas above, and from direct handling of the *Free*1 invariant itself. For *NEW*1 the (almost 16) non-linear equations in the proof were okay,

but for *DISPOSE*1, this was clearly unmanageable. We need to rethink the invariant of *Free*1 using clearer abstractions for the predicates, even if with the same formulae: it was a matter of packing up the concepts a bit more. This motivated the final version of the Z/EVES development.

4.2.3 Z/EVES v2 — invariant packaging and abstraction

In this final Z/EVES development, we did two new things: the packaging up of invariants more clearly, and the addition of key sanity checks. There were barely no changes to *NEW*0, and most changes to the invariant were guided to improve the clarity of what was being modelled by the *DISPOSE*1 postcondition.

Sanity checks

We wrote the following sanity checks for the model so far after we realised there was a bug in one part of the development as a result of a typo. The typo enabled all proofs to go through correctly, but they were for the wrong model! The key sanity check we added was that *NEW* followed by *DISPOSE* under both levels would lead to the identity. Upon failing this proof, the typo(s) in the model involving things like $+1$ or $\leq$ errors (instead of $+0$ and $<$).

Inventing new concepts

Firstly, let us remember the original invariant:

$$\begin{array}{l} \mathit{Free}1 = \mathit{Piece}\text{-}\mathbf{set} \\ \mathbf{inv}\ (\mathit{ps}) \triangleq \quad \forall p1, p2 \in \mathit{ps} \cdot \\ \qquad\qquad (p1 = p2 \lor \mathit{locs\_of}(p1) \cap \mathit{locs\_of}(p2) = \{\,\} \\ \qquad\qquad \land\ \mathit{LOC}(p1) + \mathit{SIZE}(p1) \neq \mathit{LOC}(p2)) \end{array}$$

It states different *Pieces* must have disjoint and non-abutting locations. Technically, provers tend to normalise terms, so the disjunction above actually appears in goals as

$$\begin{array}{l} \forall p1, p2 \in \mathit{ps} \cdot \\ \qquad (p1 \neq p2) \ \Rightarrow\ \mathit{locs\_of}(p1) \cap \mathit{locs\_of}(p2) = \{\,\} \\ \qquad\qquad\qquad \land\ \mathit{LOC}(p1) + \mathit{SIZE}(p1) \neq \mathit{LOC}(p2) \end{array}$$

For the invariant, we created the following (new, organising) concepts about the invariant; they can be given in VDM as $\mathbb{B}$-valued functions, which in Z appear as just sets. They were:

$$\begin{array}{l} \mathbf{unique}\ \_ == \{\mathit{fr}\colon \mathit{Piece}\text{-}\mathbf{set} \mid \forall p1, p2 \in \mathit{fr} \cdot \mathit{LOC}(p1) = \mathit{LOC}(p2) \ \Rightarrow\ p1 = p2\} \\ \_\ \mathbf{before}\ \_ == \{p1, p2 \in \mathit{Piece} \mid \mathit{LOC}(p1) + \mathit{SIZE}(p1) < \mathit{LOC}(p2)\} \\ \mathbf{sep}\ \_ == \{\mathit{fr}\colon \mathit{Piece}\text{-}\mathbf{set} \mid \forall p1, p2 \in \mathit{fr} \cdot \mathit{LOC}(p1) < \mathit{LOC}(p2) \ \Rightarrow\ p1\ \mathbf{before}\ p2\} \\ \mathit{inv}\text{-}\mathit{Free}1 == \{\mathit{fr}\colon \mathit{Piece}\text{-}\mathbf{set} \mid \mathbf{sep}\,(\mathit{fr}) \land\ \mathbf{unique}\ (\mathit{fr})\} \end{array}$$

The original invariant has too weak an invariant for uniqueness of *Piece* ($p1 = p2$). What really matters is that their locations are unique, rather than the whole Piece (*i.e.* $\mathit{mk}\text{-}\mathit{Piece}(0,5) \neq \mathit{mk}\text{-}\mathit{Piece}(0,3)$, yet sharing the same location is undesirable). Instead, first we define uniqueness (**unique**) with respect to piece's location! Next, to avoid non-linear equations around, we wrap up non-abuttingness by creating the concept of a *Piece* coming **before** another by stating their locations are apart beyond just the *SIZE*. Next, we generalise this notion to a whole set of *Piece* and call it separateness between all pieces of a set. Finally, the new invariant for *Free*1 is defined in terms of a set of *Piece* where all elements have unique locations and are explicitly separate.

The notion of separation and before survived and is used in Section 2.2. For uniqueness, the next step was to think up a better data type representation that documented the design decision of location uniqueness more clearly. The obvious solution is to use a function from

$Loc \xrightarrow{m} \mathbb{N}_1$, where each mapping represent a unique *Piece*. This would lead to necessary changes to auxiliary functions *locs-of* and *locs*, as well as to the new concept of **before** and **sep**.

New concepts for *DISPOSE*1 postcondition

The definition of *NEW*1 carried through from Z/EVES v1 with the slight adjustment about *Piece* sizes being $\mathbb{N}_1$ (*i.e.* it carried a disjunction over each case for equal and greater than, as featured in Section 2.2), but *DISPOSE*1 also needed new concepts to become both clearer and without the need to refer to *locs* in the postcondition. Like with *NEW*1, we needed to declare its behaviour explicitly.

We toyed around with two concepts:

$$\_ \textbf{ wellplaced } \_ == \{p1, p2 \in Piece \mid \textbf{ unique } (\{p1, p2\}) \land (p1 \textbf{ before } p2 \lor p2 \textbf{ before } p1)\}$$

$$\_ \textbf{ fuse } \_ == \{p1, p2 \in Piece \mid LOC(p1) + SIZE(p1) = LOC(p2)\}$$

They we were useful for proofs of lemmas involving separability, and in the old definition of *DISPOSE*1 for Z/EVES v1 of the model. **fuse** in particular, features in the final definition of *DISPOSE*1 given in Section 2.2. Furthermore, we also added the notion of abutting pieces explicitly, again as a (VDM) $\mathbb{B}$-valued function represented as a set (in Z).

$$\_ \textbf{ abutt } \_ == \{p1, p2 \in Piece \mid p1 \textbf{ fuse } p2 \lor p2 \textbf{ fuse } p1\}$$

The new definition of *NEW*1 and *DISPOSE*1 are given as follows. Note the interface change that requested sizes cannot be zero. Also for *NEW*1, instead of leaving the implicit choice in the post condition as

$$f_1 = (\overleftarrow{f_1} - \{p\}) \cup (locs\text{-}of(p) - locs\text{-}of(res))$$

covering both cases of equal and greater than, we make it explicit with implications instead. We also made a specific design decision for choosing the new location to be within the rightmost part of the (possibly larger) peace. We could have made this more non-deterministic by arbitrary choosing either (left or right most) side. Curiously, when translating the models to Isabelle, we missed this issues and made a mistaken implementation of *NEW*1, as described below in the next Sections 4.2.4 onwards

*NEW*1 $(req: \mathbb{N}_1)$ $res: Piece$

ext wr f_1 : *Free*1

pre $\exists r \in f_1 \cdot SIZE(r) \geq req$

post $\exists p \in Piece \cdot p \in \overleftarrow{f_1} \land$
$SIZE(p) \geq req \land$
$res = mk\text{-}Piece(LOC(p), req) \land$
$SIZE(p) = req \Rightarrow f_1 = (\overleftarrow{f_1} - \{p\}) \land$
$SIZE(p) > req$
$\Rightarrow f_1 = (\overleftarrow{f_1} - \{p\}) \cup \{mk\text{-}Piece(LOC(p) + req, SIZE(p)\text{-}req)\}$

*DISPOSE*1 $(ret: Piece)$

ext wr f_1 : *Free*1

pre $locs\_of(ret) \cap locs(f_1) = \{\}$

post $\exists join, abut \in Piece\textbf{-set} \cdot$
$abt = \{q \in Piece \mid q \in f_1 \land ret \textbf{ abutt } q\} \land$
$join = \{ret\} \cup abt \land$
$f_1 = (\overleftarrow{f_1} - join) \cup \{mk\text{-}Piece(min\text{-}loc(join), sum\text{-}size(join))\}$

The definition of *DISPOSE*1 postcondition now explicitly declares how the locations are affected as a result of returning memory to *Free*1. The stat update first remove a larger set of joined pieces composed of any (possibly) abutting pieces to the one returned, together with the piece being returned. In the best case, nothing abuts, and the join is removed to be added straight after. If any piece locations abut, then a calculation is made to pick out the minimal location with the summed sizes of involved pieces as the new (larger) piece returned to *Free*1. The abutting set is defined as any piece within the before state ($q \in \overleftarrow{f_1}$) that **abutt** ts. The new **abutt** concept ensures that the involved pieces **fuse** at either side, which entails specific alignment of locations as defined by the new concepts given above.

Lemmas about the new concepts

This process led to some new lemmas involving the novel concepts that were useful in understanding the problems within proofs. For instance, we proved these lemmas about **before** and **unique** that were useful

lemma lPieceExcludedMiddle
$\forall p, q \in Piece \cdot$
p **before** $q \Rightarrow \neg\, q$ **before** p

lemma lFree1UniqueUnion
$\forall f \in Free1, p \in Piece \cdot$
unique $(f \cup \{p\})$
$\Leftrightarrow$
$(\forall q \in f \cdot LOC(q) = LOC(p) \Rightarrow SIZE(q) = SIZE(p))$

The first lemma captures the asymmetry between abutting pieces (below and above) that we were trying to get a symmetric description of. This was an attempt at too strong a simplification to the concepts that led to more confusing outcome (*i.e.* we needed both the concepts of before and after pieces).

Finally, now with the notion of uniqueness clearly stated for the new version of the *Free*1 invariant, we managed to prove what are the conditions for extending *Free*1 under union. which we had failed before (see Lemma lFree1UnitUnion above). The notion of **abutt** was not ideal, though. It kept a hidden case analysis on **fuse** on either side that was unhelpful.

With this we finalise our historical reconstruction of the problem of developing the heap through a Z theorem prover in order to discharge both satisfiability and refinement proofs.

4.2.4 Isabelle v0 — warts and all

This first Isabelle encoding of the problem faced more challenging issues regarding the problem representation because of a significant difference in the Logic (HOL) and the type system (*i.e.* no explicit support for dependant types). In itself this is interesting, as it highlights the pitfalls of using non-native theorem provers for discharging formal proofs.

For instance, in Isabelle, if we want to define VDM's $\mathbb{N}_1$ as just a subset of Isabelle's nat it is not adequate. That is because our subtype cannot be used as part of any other type declaration or in signature of functions. The appropriate way would be to instantiate our own encoding of $\mathbb{N}_1$ to the corresponding type classes representing commutative mono ids, so that we would enjoy all the Isabelle machinery for non-linear arithmetic and natural number induction.

We also worked out ways to use Isabelle's locale to keep track of underlying type invariants, type assumptions, and preconditions, which we needed to record explicit everywhere needed. In fact, we missed the type invariant in a few satisfiability proof obligations. To our surprise when the proof went through easily, we were readily suspicious something had gone wrong in the quick and dirty translation. This led to a more systematic approach, as the one described later in Chapter 5.

We basically had two versions within this bracket, that mimicked the updates / evolutions discussed above for the Z/EVES theorem prover. That was until we got to the use of VDM maps within Isabelle's own type system, where we needed to carefully rethink our whole strategy.

Using Isabelle maps was fruitful yet not entirely satisfactory, and it became clear a VDM library for map operations would be necessary. For instance, we needed to explicitly define VDM map union as well as domain subtraction (or anti-restriction). All this was not needed in Z/EVES given the Z mathematical toolkit is already quite close to VDM's.

4.2.5 Isabelle v1 — Sledgabelle lemmas

This version included $\mathbb{B}$-valued functions to represent type restricting predicates, we introduced a basic VDM maps library, and a structured locale hierarchy for the heap hypothesis. Moreover, we started structuring the specification according to VDM's pre defined functions for each operation. That is, we defined functions pre, post, and invariant for each involved operation and restricted type. One key aspect is to ensure the type invariant is kept in the after state when defining post conditions, which we had missed initially.

Some lemmas from the Z/EVES development involving *locs-of* were translated, and we started making extensive use of Isabelle's sledgehammer tool, an automatic proof finder that makes use of various SAT/SMT solvers. We started dividing and structuring lemmas in small enough chunks, such that sledgehammer would find the proofs for them. Thus, the proof of $NEW1$ postcondition in Isabelle became a matter of slicing the goal in small enough chunks for sledgehammer to smash them away. Up to the feasibility proof of $DISPOSE1$, we had almost 2/3 of lemmas "sledgehammerable".

Many such lemmas were not quite general, but rather intermediary steps in a larger proof. Nevertheless, this "strategy" proved effective in resolving the easier proofs within the heap proof obligations. It also highlighted the most effective way to shape lemmas for Isabelle's simplifier to use. Such lemma shaping is crucial, and quite different from the way on would shape lemmas in Z/EVES. This highlights the some key differences between both provers used in the problem.

Modelling map comprehension for partial VDM maps

For the proof of $DISPOSE1$, this strategy was not going to work so well. That is because VDM map representation in Isabelle was tricker than Z/EVES: in Isabelle all functions are total, whereas in Z/EVES partial functions are common. Isabelle handles partiality (in their implementation of VDM maps) as a function to an optional type[3].

At first we used λ-abstractions to represent map comprehension and the various map operators. This proved to be a ill-chosen representation, as most of the machinery available to handle Isabelle maps were not prescribed for such choice. This made it clear for us that in order to effectively use Isabelle, we would need to model according to the prescribed choices in the Isabelle's libraries we were using.

Another key problem in $DISPOSE1$ was to represent map comprehension, which was hard to write in Isabelle (for us). Instead, we kept the λ-abstractions isolated to the map operators like domain subtraction, and tried to use set theory (and set comprehension) for needed definitions, instead. This solved the problem and enable us to progress with our proofs.

Finally, after discovering mistakes in the translation, we realised a more systematic (if still informal) approach was needed. We created a set of translation templates to ensure that naming conventions (and variable capture within the locale) were not producing the wrong models in Isabelle. Added sanity checks ensured that the translation was as good as it was going to get without mechanised assistance.

[3]Arguably, there are alternative approaches to handling partial functions. Using Isabelle's Map.thy library was our choice.

At this stage, for the proof of *DISPOSE*1 satisfiability (and later refinement between level 0 and 1), proofs became quite large and laborious. At this stage, we decided to review the whole development and start afresh, now we have understood the problem well.

Once we had the models described (as explained in the next Chapter 5), we decided to fork our proof development in two parts side-by-side: one using procedural Isabelle proofs, and the other using declarative Isar proofs; both of which we would use our *ProofProcess* tool [Vel12] (see Section 7.2) to capture data about the proof attempts. As we had already collected such data for the Z/EVES development, we wanted to compare the data within (two independent) Isabelle developments as well. The proof process data is subset of what is described in Chapter 3.

4.3 Summary

In this Chapter we presented a brief summary of the development history of our heap models using both Z/EVES and Isabelle. In the end, we favoured the Isabelle implementation for further discussion for various reasons.

One, Isabelle is a more general and widely used theorem prover, and it is also within the remit of AI tools developed by our partners within AI4FM. Second, Isabelle has more powerful tools to aid proof description and discovery. Having said that, the representation of the heap in Z/EVES was easier and more natural, given VDM is closer to Z than to HOL!

In coming Chapters we describe the details of our final and complete formalisation of levels 0 and 1 in Isabelle, including satisfiability, refinement, and sanity checks.

We also use Z/EVES and Isabelle development as a way of collecting proof process information. Our Eclipse-based tool for capturing the proof process was used to collect data to inform the development of the meta-model described in Chapter 3.

Chapter 5

Heap in Isabelle

5.1 Introduction

This chapter and the next continue the exposition of the heap storage case study by describing the formalisation and formal verification in the Isabelle proof assistant [NPW02b] of the latest heap model presented in Chapter 2.

In the next section (Section 5.2), we briefly introduce the Isabelle proof assistant and its proof languages. Then, in Section 5.2.4 we give a general description of how VDM operations and functions are formalised in Isabelle, giving details of the important differences. Section 5.3 presents the Isabelle models for level 0 and level 1. The latex code that presents these models is directly generated from the proof developement. This section is paired with Appendix B, which details our naming and stylistic conventions in the formalisation.

Chapter 6 describes the proof obligations and provides a broad overview of the formal verification. We pursued two parallel verification efforts in Isabelle: Freitas, using a procedural style of proof, leveraging Isabelle's automation; Whiteside used the declarative Isar language. We provide a broad comparison of the two proof efforts. The full proofs can be found on in Appendices F and E.

5.2 The Isabelle proof assistant

Isabelle is a generic theorem prover or, rather, a logical framework with a meta-logic called Isabelle/Pure (minimal intuitionistic higher order logic) in which object logics are encoded. We use the most popular, and best supported, object logic: classical higher order logic (referred to as Isabelle/HOL).

In this section, we describe the elements of Isabelle required to understand the rest of this technical report. Section 5.2.1 details the proof languages used by Freitas and Whiteside, providing a brief comparison of their features. Then Section 5.2.2 introduces the Sledgehammer and Nitpick tools which are important in harnessing automation and checking for counterexamples, respectively. Section 5.2.4 introduces our VDM library and highlights three key differences between the Isabelle representations and the VDM logic that are important to understand the formalisation that follows. Finally, we summarise in Section 5.2.5.

5.2.1 Isabelle proof languages

The core proof language for Isabelle is called Isar [Wen02]. Broadly speaking, it permits two styles of proof: declarative, where the state of the proof is encoded in the proof script; and,

procedural, where the state of the proof can only be seen upon replay. As a simple illustration, we give two proofs in Isabelle using each style. The proof shown is part of the proof of commutativity of addition for natural numbers.

```
theorem natcom-procedural:
(a::nat) + b = b+a
apply (induct a)
apply (subst add-0)
apply (subst add-0-right)
apply (rule refl)
sorry

theorem natcom-dec: (a::nat) + b = b+a
proof (induct a)
  show 0 + b = b + 0
  proof -
    have 0+b=b by (simp)
    also have ...=b+0 by (simp)
    finally show ?thesis .
  qed
 next
  fix a
  assume in-hyp: a + b = b + a
  show Suc a + b = b + Suc a
    sorry
qed
```

As can be seen, the prodedural style is more compact, but it is not clear without re-running the proof what the goals being operated on are. Furthermore, it is difficult to see the branching structure of the proof because of the linear structure and the fact that some tactics apply to just a single subgoal, while others apply to several.

The declarative style is longer, but can be read without needing to run the system; furthermore, it enables a natural forwards style of proof that is closer to normal mathematical practice. For a more detailed comparison of both styles of proof, Harrison's 'Proof Style' is recommended [Har96].

5.2.2 Sledgehammer and Nitpick

Isabelle also has two important external tools that have been used extensively in this project: Sledgehammer [PB10] and Nitpick [BN09].

5.2.2.1 Sledgehammer

Sledgehammer is a tool to find automatic proofs of goals. Invoking sledgehammer will send the current goal to multiple automated theorem provers, like Z3, Vampire, Spass, *etc* along with a set of lemmas from the library that sledgehammer "thinks"[1] will be useful. If one of the ATPs succeeds, then it can be translated to an Isabelle proof, using a tactic called 'metis'. As a simple example, the following lemma (a lemma from the VDM maps library) has been proved automatically by sledgehammer, and requires three lemmas (facts) to be passed to metis.

```
lemma metis-example:
  assumes ∗: x ∉ dom f
  shows  x ∉ dom (s -◁ f)
```

[1] The selection process/criteria here is itself interesting and worth further investigation. How does sledgehammer know what to use/filter?

by (*metis* ∗ *domIff dom-antirestr-def*)

Sledgehammer can be more powerful than Isabelle's automated tactics (such as simp and auto) on domain reasoning because it can automatically select the appropriate lemmas to use, rather than performing time-consuming configuration of the simplifier. However, it can fail in domains where Isabelle has been finely tuned, such as sets, since there are many potential lemmas that can be selected.

5.2.2.2 Nitpick

Nitpick is a powerful counterexample checker for Isabelle and can be invoked to check the validity of the lemma you are attempting to prove. For example, running nitpick on the lemma above without the assumption ∗:

lemma *nitpick-example*:
shows $x \notin dom\ (s \mathbin{-\!\triangleleft} f)$
nitpick

gives the following counterexample: $f = [a_1 \mapsto b_1,\ a_2 \mapsto b_1]$, $s = \{a_2\}$, and $x = a_1$, which makes clear the issue with the current conjecture.

5.2.3 Proof styles

In this section, we elaborate a little on the top-level proof styles (patterns) used by Whiteside and Freitas.

5.2.3.1 Proof sketches - Whiteside

The general method for proof used by Whiteside is akin to Wiedjik's formal proof sketches [Wie02]. The main idea is to write all the main proofs in a declarative style and start with a rough sketch and gradually fill it in. To construct the proof sketch, Whiteside has in mind how the proof should go (either from intuition or a pencil and paper version) and writes out the main steps (using the *sorry* command to omit the proof) . Then, the main steps should be combined to solve the goal using the default automation of Isabelle. For example, a proof of a subgoal (that occurs in a few places) could be sketched as follows:

have *disjoint* (*locs-of* (*l* + *s*) (*the* (*f l*) - *s*)) (*locs* ({*l*} -◁ *f*))
proof -
have (*locs-of* (*l* + *s*) (*the* (*f l*) - *s*)) ⊆ *locs-of l* (*the* (*f l*))
sorry
moreover have *disjoint* (*locs-of l* (*the* (*f l*))) (*locs* ({*l*} -◁ *f*))
sorry
ultimately show *?thesis* **by** *auto*
qed

This type of sketch is called a combinatory sketch, because all the facts introduced are combined to solve the goal using the isabelle auto tactic. From inspection, it is clear that the sketched facts are enough to give the gist of the proof: to show $A \cap B = \{\}$, we note that $A \subseteq A'$ and that $A' \cap B = \{\}$ (recall that two sets are *disjoint* if their intersection is empty).

It is important to note that the proofs of the sketched elements may be arbitrarily complicated and will often be solved with further sketches, but they may also be solved by automation. The advantages of the sketching pattern is that it provides a clear route through the proof from the outset; a disadvantage is that the 'clear route' may lead up a blind alley if the, e.g., nth step is not valid and a lot of wasted time is spent on the n-1 prior steps. In practice this doesn't occur much and when it does, the n-1 are usually useful in a revised sketch.

5.2.3.2 Sledgabelle - Freitas

This is as discussed in Section 4.2.5.

5.2.4 VDM library

Our model of the heap is built upon the core VDM datatypes and operators: natural numbers, positive natural numbers, sets, and (finite, partial) maps. The Isabelle/HOL library already supports most of these concepts, but in some cases we needed to define further operators. We needed to define domain subtraction (or antirestriction) on maps, for example:

$s \text{ -}\triangleleft\ m \equiv \lambda x.\ \mathit{if}\ x \in s\ \mathit{then}\ \mathit{None}\ \mathit{else}\ m\ x$

and proved associated lemmas that would be considered part of a VDM Library, such as the domain of an anti-restricted map:

$\mathit{dom}\ (S \text{ -}\triangleleft\ f) = \mathit{dom}\ f \text{ - } S$

which links some map operators to set operators. The table in Figure 5.1 gives an overview of the VDM library. Each operator is shown, alongside its syntax, with the number of lemmas about it (as the root of the term tree) and the number of times that lemmas about this operator were used in both proof developments[2].

Operator	**Symbol**	**Number Lemmas**	**Freitas Total**	**Whiteside Total**
Domain restriction	$\triangleleft$	15	15	28
Domain anti-restriction	$\text{-}\triangleleft$	23	80	61
Map override	$\dagger$	22	54	20
Map union	$\cup m$	24	71	39
Total		92	220	148

Figure 5.1: The VDM Library in Isabelle

We note three important differences between VDM and the representation in Isabelle/HOL:

1. Isabelle support partial functions is involved/limited, and not a basic concept, like Z's set of pairs of VDM's primitive (partial maps) type. Thus, the partiality of maps is achieved using the *option* datatype. Thus, elements of the map are accessed using the special *the* operator, for example:

   $\{x\} \triangleleft f = [x \mapsto \mathit{the}\ (f\ x)]$

   describes the result of domain restriction on a singleton set (under the assumption $x \in \mathit{dom}\ f$). *the* operator is used for accessing an actual value within a map. That is, the domain element is known and we have a value. When map application happens on an element outside the domain, Isabelle returns *None*, a bottom element that totalises VDM maps in Isabelle.

2. Secondly, maps (and sets) are not necessarily finite. Thus, lemmas about finiteness of composite maps are required, for example:

   $\mathit{finite}\ (\mathit{dom}\ (f \cup m\ g))$

[2] Approximately.

if *finite* (*dom f*) and *finite* (*dom g*).

3. Finally, there is no $\mathbb{N}_1$ datatype in Isabelle. To get around this, we define a predicate *nat1* and extend it to operate on sets and maps (see Section 5.3.2 for the definitions on sets and maps).

definition
nat1 :: *nat* ⇒ *bool*
where
nat1 n ≡ *n* > *0*

To make $\mathbb{N}_1$ a type with access to non-linear arithmetic operators and automation, one needs to instantiate that new type to various type classes, hence effectively create an algebra for $\mathbb{N}_1$!

There is an important difference between the finiteness requirement and the $\mathbb{N}_1$ requirement. The finiteness is not part of the heap model, per se, but required as preconditions for many standard Isabelle lemmas that we need (defining *sum-size*, for instance)[3]. On the other hand, $\mathbb{N}_1$ is very much a part of the model; this means that we need to keep track of the VDM $\mathbb{N}_1$ type by introducing predicates in many places, resulting in a slightly messy specification and conditional VDM functions, such as:

definition
locs-of :: *Loc* ⇒ *nat* ⇒ (*Loc set*)
where
locs-of l n ≡ (*if nat1 n then* { *i*. *i* ≥ *l* ∧ *i* < (*l* + *n*) } *else undefined*)

which would not be required if we could specify:

definition
locs-of-nat1 :: *Loc* ⇒ *nat1* ⇒ (*Loc set*)
where
locs-of-nat1 l n ≡ { *i*. *i* ≥ *l* ∧ *i* < (*l* + *n*) }

Using this definition, we would need to instantiate *nat1* through various type classes in Isabelle, which was beyond what we wanted to do.

These conditions add to the complexity of the proof somewhat, but we use Isabelle's automation to reduce the burden considerably. The remaining effort is managable (both in terms of proof effort and effort ensuring the model is correct) in a project of comparable size to the heap. However, we expect that proper support for the VDM datatypes would be required for any larger model verification.

5.2.5 Summary

This section has introduced Isabelle, its proof languages, and tools for improving automation and counterexample checking. We also discussed the VDM library that we built as part of the heap case study. This library represents a considerable chunk of our proof effort (about 20%) and was used extensively throughout the heap verification. Fortunately, these results are transferable to any other VDM model verification[4]. We have not yet built in any automation support—simplifier sets for example—for the library as of yet. In the heap case study, all lemmas were explicitly specified when used, leading to a larger proof, but with explicit dataflow which allowed us to collect some statistics about the proofs. For a concrete framework for VDM verification, finely tuned automation would considerably ease the burden of proof.

[3] In VDM all sets (and maps) are finite by definition.

[4] Though, we note that this library is expected to grow slightly as lemmas that we missed the first time round suggest themselves, and because we only cover a few of the available VDM map operators

Finally, in this section, we detailed the three main differences between Isabelle and VDM and our (or Isabelle's) techniques for bridging the difference. Again, for proper support for VDM verification in Isabelle, more permanaent support for the VDM datatypes, such as $\mathbb{N}_1$ and partial maps would be required, but that is beyond the scope of this project.

5.3 The models in Isabelle

We now turn to the actual model of the heap as specified in Isabelle. The next section details Level 0 (Section 5.3.1), and Section 5.3.2 details Level 1, as presented in Chapter 2. The justifications for the formalisation are given when they are first introduced, and Section 5.3.3 summarises the general transformation strategy. A detailed account of our naming conventions is provided in Appendix B.

5.3.1 Heap level 0

In analogy with the VDM specification (see Section 2.1), we first define some type synonyms to represent locations and the state:

type-synonym *Loc′ = nat*
type-synonym *F0′ = Loc′ set*

The auxiliary definitions of *locs-of* (shown above) and *is-block* can then be defined with appropriate guards on any instances of the $\mathbb{N}_1$ type in VDM.

definition *is-block :: Loc ⇒ nat ⇒ (Loc set) ⇒ bool*
where
is-block l n ls ≡ nat1 n ∧ locs-of l n ⊆ ls

The next step in specifying the model is to create definitions for the invariant, preconditions, and post-conditions for each operation. We encode the finiteness requirement in Isabelle as an invariant on level 0 (note that this doesn't exist and is not required, since all sets are finite in VDM).

definition
F0-inv :: F0 ⇒ bool
where
F0-inv f ≡ finite f

definition
new0-pre :: F0 ⇒ nat ⇒ bool
where
new0-pre f s ≡ (∃· l. (is-block l s f))

definition
new0-post :: F0 ⇒ nat ⇒ F0 ⇒ Loc ⇒ bool
where
new0-post f s f′ r ≡ (is-block r s f) ∧ f′ = f - (locs-of r s)

definition
dispose0-pre :: F0 ⇒ Loc ⇒ nat ⇒ bool
where
dispose0-pre f d s ≡ locs-of d s ∩ f = {}

definition
dispose0-post :: F0 ⇒ Loc ⇒ nat ⇒ F0 ⇒ bool
where

dispose0-post f d s f' ≡ f' = f ∪ locs-of d s

As can be seen, the definitions are identical to the VDM specification, except that these definitions require all parameters to be explicitly provided. We now encode variants of the pre and postconditions where the inputs and state are implicit using locales. They make for an Isabelle theory that is closer to the VDM model and is also less repetitive.

VDM operations are defined using locales to keep hold of the state and its invariant as part of the locale assumptions, and similarly for inputs. Locales provide a uniform technique for packagaing together a VDM 'operation'. The encoding is not perfect, however, because post-conditions need to be specified separately (though, within the locale context).

We use layered locales to avoid repetition of the state invariant across each operation of interest and to provide a natural context for the adaquecy proof (which is independent of the individual operations).

locale *level0-basic* =
  fixes *f0* :: *F0*
  and *s0* :: *nat*
 assumes *l0-input-notempty-def*: *nat1 s0*
  and *l0-invariant-def* : *F0-inv f0*

In *level0-basic*, we introduce the state *f0* and an input *s0*, which corresponds to the size of the heap memory required to be allocated or disposed. Then, we ensure that the size is non-zero with a locale assumption (corresponding to the type in VDM) and the invariant representing finiteness. We consider *l0-input-notempty-def* as an assumption because it is a property of the input; the finiteness is an invariant because it is defined over the state.

The actual VDM operations are then defined by locale extension and a definition for the postcondition:

locale *level0-new* = *level0-basic* +
  assumes *l0-new0-precondition-def*: *new0-pre f0 s0*

definition (**in** *level0-new*)
 new0-postcondition :: *F0* ⇒ *nat* ⇒ *bool*
where
 new0-postcondition f' r ≡ *new0-post f0 s0 f' r* ∧ *F0-inv f'*

The locale *level0-new* extends the locale *level0-basic* with the precondition, where the parameters have been supplied by the fixed variables for this level. Note there is no need to check the invariant for *f0* at the *new0-precondition*, since it is already stated as a locale assumption at *level0-basic*. The postcondition *new0-postcondition* is then specified in the context of the *level0-new* (meaning all the fixed variables are available) and is defined to take two parameters:

1. The updated state *f'*;
2. and, the result *r* that represents the start location for the allocated block.

These two parameters are the variables to be existentially quantified when proving satisfiability (a.k.a feasibility) proofs for *NEW*. The definition consists of a conjunction of the *new0-post* definition, with the appropriate parameters instantiated, and the invariant predicate on the updated state. Note that an updated invariant condition is necessary and is hidden in a VDM operation specification (and appears when POs are generated, by Overture[5], for example), but must be manually added in Isabelle.

The dispose operation is similarly defined, additionally requiring an extra input variable: the start location *d0* of the block the add back to the heap, as in Chapter 2.

locale *level0-dispose* = *level0-basic* +

[5]See `http://www.overturetool.org`

fixes *d0* :: *Loc*
assumes *l0-dispose0-precondition-def*: *dispose0-pre f0 d0 s0*

definition (**in** *level0-dispose*)
dispose0-postcondition :: *F0* ⇒ *bool*
where
dispose0-postcondition f′ ≡ *dispose0-post f0 d0 s0 f′* ∧ *F0-inv f′*

Given totalisation and definedness of the VDM model here, only feasibility proof obligations per level are needed. These are also given as definitions within the locale (where the fixed variables can be seen as universally quantified, and assumptions can be seen as assumption of the theorem).

definition (**in** *level0-new*)
PO-new0-feasibility :: *bool*
where
PO-new0-feasibility ≡ (∃· *f′ r′* . *new0-postcondition f′ r′*)

definition (**in** *level0-dispose*)
PO-dispose0-feasibility :: *bool*
where
PO-dispose0-feasibility ≡ (∃· *f′* . *dispose0-postcondition f′*)

These PO definitions are the top-level goals to be discharged using Isabelle. We provide more details of the proof obligations in Chapter 6.

Finally, it is worth explaining that within the locale structure, we are actually proving the usual proof obligation setup, which would be more familiar if given outside the locale as:

definition
PO-new0-fsb :: *bool*
where
PO-new0-fsb ≡ (∀· *f s* . *F0-inv f* ∧ *nat1 s* ∧ *new0-pre f s* ⟶
(∃· *f′ r′* . *new0-post f s f′ r′* ∧ *F0-inv f′*))

definition
PO-dispose0-fsb :: *bool*
where
PO-dispose0-fsb ≡ (∀· *f d s* . *F0-inv f* ∧ *nat1 s* ∧ *dispose0-pre f d s* ⟶
(∃· *f′* . *dispose0-post f d s f′* ∧ *F0-inv f′*))

The locale based definitions are implied by the generic version, which universally quantify what is localy assumed.

These locale-based PO definitions are the top-level goals to be discharged using Isabelle. We provide more details of the proof obligations in Chapter 6.

5.3.2 Heap level 1

Firstly, we define a type type synonym for the state of the free store at level 1 to be a map from locations to sizes:

type-synonym *F1* = *Loc* ⇀ *nat*

5.3.2.1 Auxillary functions

Note that the size is only *nat* here so, as mentioned earlier, we must extend the *nat1* predicate to operate on maps and sets to ensure that the model is consistent with VDM:

definition

```
  nat1-map :: F1 ⇒ bool
where
  nat1-map f ≡ (∀· x. x ∈ dom f ⟶ nat1 (the (f x)))

definition
  nat1-set :: (nat set) ⇒ bool
where
  nat1-set S ≡ (∀· x. x ∈ S ⟶ nat1 x)
```

The level 1 model introduces a new auxiliary function, *locs* that returns the set of all free locations withing a given map. We define the *locs* function using a union over the elements in the domain of the VDM map. It is wrapped inside a conditional expression, however, in order to ensure that the map is appropriately a *nat1-map*:

```
definition
  locs :: (Loc ⇀ nat) ⇒ Loc set
where
  locs sm ≡ (if nat1-map sm then
               ⋃ s ∈ dom sm. locs-of s (the (sm s))
             else
               undefined)
```

It is otherwise *undefined*, which is a polymorphic constant in Isabelle. That is, the VDM model uses a total map to $\mathbb{N}_1$, whereas here we can only use a map to $\mathbb{N}$ as a parameter. Thus, we totalise the definition of *locs* by giving it a bottom element (as Isabelle's *undefined*) when the expected type fails.

It is important to emphasise this is not VDM's notion of undefinedness. For instance, it is possible to prove that *undefined* = *undefined* in Isabelle, which is not true in VDM's three-valued logic. Thus, *undefined* should never feature in our proofs. If it does, it means we made some mistake somewhere by applying a function to the wrong type. For further discussion on the subtleties of handling partial functions, see [Jon95, Sch12].

5.3.2.2 Invariant

Recall the level 1 invariant in Section 2.2:

$$
\begin{array}{l}
\mathit{Free}1 = \mathit{Loc} \xrightarrow{m} \mathbb{N}_1 \\
\textbf{inv } (f) \triangleq \\
\quad \forall l, l' \in \textbf{dom } f \cdot \\
\qquad l \neq l' \;\Rightarrow\; \mathit{is\text{-}disj}(\mathit{locs\text{-}of}(l, f(l)), \mathit{locs\text{-}of}(l', f(l'))) \wedge \\
\quad \forall l \in \textbf{dom } f \cdot (l + f(l)) \notin \textbf{dom } f
\end{array}
$$

It contains two components (a conjunction):

- *Disjoint*: that the locations defined by each element in the map are disjoint;
- and, *sep*: that the locations defined by elements do not abut on any end.

We encode these as individual definitions in Isabelle:

```
definition
  Disjoint :: F1 ⇒ bool
where
 Disjoint f ≡
     (∀· a ∈ dom f. ∀· b ∈ dom f . a ≠ b ⟶ disjoint (Locs-of f a) (Locs-of f b))

definition
  sep :: F1 ⇒ bool
```

where
sep f ≡ (∀ · *l* ∈ *dom f* . *l* + *the*(*f l*) ∉ *dom f*)

where *disjoint A B* is the same as $A \cap B = \{\}$, and *Locs-of f a* is the same as *locs-of a* (*the* (*f a*)).

Albeit trivial, this decomposition into separate concepts is invaluable in taming the goal complexity during proofs (see discussion in Section 4.2). They create what we call "zoom" levels of interest/discourse. For instance, we create various lemmas about these definitions and their relationship with, say *locs-of* and *locs* or set theory and map operators. So, in actual POs, these issues of mechanisation are already distilled and resolved.

We must also, however, have additional components to the invariant. They are the implicit VDM notion of finiteness of maps and sets, and the subtype checking on map range type for $\mathbb{N}_1$.

- *nat1_map*: that the state doesn't contain any locations that map to size 0.
- *finite domain*: that the domain of the map is finite, similarly to level 0 state.

Thus, the invariant definition is as follows:

definition
F1-inv :: *F1* ⇒ *bool*
where *F1-inv f* ≡ *Disjoint f* ∧ *sep f* ∧ *nat1-map f* ∧ *finite*(*dom f*)

definition
VDM-F1-inv :: *F1* ⇒ *bool*
where
VDM-F1-inv f ≡ *Disjoint f* ∧ *sep f*

We also define the VDM invariant, as we may wish to discharge the Isabelle parts the invariant first (finiteness etc), as they are often simpler. We provide a lemma to 'shape' the goal as such:

lemma *invF1-shape*: *nat1-map f* ⟹ *finite* (*dom f*) ⟹ *VDM-F1-inv f* ⟹ *F1-inv f*
unfolding *F1-inv-def VDM-F1-inv-def* **by** *simp*

Such proof decomposition is again essential for automation and proof strategy reuse, as it informs (meta-)data collection (see Chapter 3 on meta-data and Chapter 6 on Isabelle proofs).

Furthermore, we define introduction and elimination rules to help unfold the invariant; we also provide weakening rules for the case that only one part of the invariant is required (we only show the *sep* version here):

lemma *invF1E*[*elim*!]: *F1-inv f* ⟹ (*sep f* ⟹ *Disjoint f* ⟹ *nat1-map f* ⟹ *finite* (*dom f*) ⟹ *R*) ⟹ *R*
unfolding *F1-inv-def* **by** *simp*

lemma *invF1I*[*intro*!]: *sep f* ⟹ *Disjoint f* ⟹ *nat1-map f* ⟹ *finite* (*dom f*) ⟹ *F1-inv f*
unfolding *F1-inv-def* **by** *simp*

lemma *invF1-sep-weaken*: *F1-inv f* ⟹ *sep f*
unfolding *F1-inv-def* **by** *simp*

5.3.2.3 NEW operation

Following the style of level 0 in Section 5.3.1, we create definitions for the pre and post-conditions for the operations. We split the NEW post-condition into two separate definitions, corresponding to each disjunct in the VDM operation. Again, this is useful for proof decomposition within POs and also to help identify hidden case analysis, another of our proof patterns.

definition
new1-pre :: *F1* ⇒ *nat* ⇒ *bool*
where
new1-pre f s ≡ (∃· *l* ∈ *dom f* . *the*(*f l*) ≥ *s*)

definition
new1-post-eq :: *F1* ⇒ *nat* ⇒ *F1* ⇒ *Loc* ⇒ *bool*
where
new1-post-eq f s f′ r ≡ *r* ∈ *dom f* ∧ *the*(*f r*) = *s* ∧ *f′* = {*r*} -⊲ *f*

definition
new1-post-gr :: *F1* ⇒ *nat* ⇒ *F1* ⇒ *Loc* ⇒ *bool*
where
new1-post-gr f s f′ r ≡ *r* ∈ *dom f* ∧ *the*(*f r*) > *s* ∧
f′ = ({*r*} -⊲ *f*) ∪*m* [*r* + *s* ↦ *the*(*f r*) - *s*]

definition
new1-post :: *F1* ⇒ *nat* ⇒ *F1* ⇒ *Loc* ⇒ *bool*
where
new1-post f s f′ r ≡ *new1-post-eq f s f′ r* ∨ *new1-post-gr f s f′ r*

5.3.2.4 DISPOSE operation

Before showing the locale definitions corresponding to the *DISPOSE*1 operation, we create auxiliary definitions for dispose. The way these came about is discussed in Section 4.2. First are the two auxilliary functions called *sum_size* and *min_loc* which are used in the postcondition are defined using Isabelle's operators for set minimal and summation, respectively.

definition
min-loc :: (*Loc* ⇀ *nat*) ⇒ *nat*
where
min-loc sm = (*if sm* ≠ *empty then*
Min (*dom sm*)
else
undefined)

definition
sum-size :: (*Loc* ⇀ *nat*) ⇒ *nat*
where
sum-size sm = (*if sm* ≠ *empty then*
(∑ *x*∈(*dom sm*) . *the* (*sm x*))
else
undefined)

Once again, we used Isabelle's *undefined* to enable a total function over a subtype, as we did for *locs*.

We have two versions of the postconditions: the exact translation from the VDM specification and a version where *above*, *below*, and *ext* are given as definitions. The latter definition makes proof more straightforward since we can refer to the maps by name and unfold where necessary. We do, of course, prove both definitions equivalent. This is another example of zooming: the use of different levels of interest in involved operators, that is based on the problem at hand, and is useful in helping proof decomposition and lemma discovery for higher automation.

definition
dispose1-pre :: *F1* ⇒ *Loc* ⇒ *nat* ⇒ *bool*
where
dispose1-pre f d s ≡ *disjoint* (*locs-of d s*) (*locs f*)

definition
  dispose1-post :: *F1* ⇒ *Loc* ⇒ *nat* ⇒ *F1* ⇒ *bool*
where
  dispose1-post f d s f′ ≡
    (∃· *below above ext* .
      below = { *x* ∈ *dom f* . *x* + *the*(*f x*) = *d* } ◁ *f* ∧
      above = { *x* ∈ *dom f* . *x* = *d* + *s* } ◁ *f* ∧
      ext = (*above* ∪*m* *below*) ∪*m* [*d* ↦ *s*] ∧
      f′ = ((*dom below* ∪ *dom above*) -◁ *f*) ∪*m* ([*min-loc*(*ext*) ↦ *sum-size*(*ext*)]))

In our alternative formulation, the three existential variables are given as definitions, for example:

definition
  dispose1-below :: *F1* ⇒ *Loc* ⇒ *F1*
where
  dispose1-below f d ≡ { *x* ∈ *dom f* . *x* + *the*(*f x*) = *d* } ◁ *f*

These encoding considerations are crucial to ensure proofs are not complicated by technicalities unrelated to the problem. One must not, however, fall for the temptation to chisel the model into whatever the theorem prover would be happier with. Our modification is clearly equivalent, and can be proved as such if that's the case, we we have done for the layered definition of dispose with respect to the original one.

The other two definitions are:

definition
  dispose1-above :: *F1* ⇒ *Loc* ⇒ *nat* ⇒ *F1*
where
  dispose1-above f d s ≡ { *x* ∈ *dom f* . *x* = *d* + *s* } ◁ *f*

definition
  dispose1-ext :: *F1* ⇒ *Loc* ⇒ *nat* ⇒ *F1*
where
  dispose1-ext f d s ≡ (*dispose1-above f d s* ∪*m* *dispose1-below f d*) ∪*m* [*d* ↦ *s*]

which allows us to write and prove:

definition
  dispose1-post2 :: *F1* ⇒ *Loc* ⇒ *nat* ⇒ *F1* ⇒ *bool*
where
  dispose1-post2 f d s f′ ≡
    (*f′* = ((*dom* (*dispose1-below f d*) ∪ *dom* (*dispose1-above f d s*)) -◁ *f*)
    ∪*m* ([*min-loc*(*dispose1-ext f d s*) ↦ *sum-size*(*dispose1-ext f d s*)]))

lemma *dispose1-equiv*:
 dispose1-post f d s f′ = *dispose1-post2 f d s f′*
unfolding *dispose1-post-defs dispose1-post2-defs*
by *auto*

5.3.2.5 VDM operation definitions and feasibility goals

Finally, we put everything together in locales and construct definitions relating to the feasibility proofs. As with level 1, we encode the shared inputs, state, assumptions and invariant in a separate locale:

locale *level1-basic* =
  fixes *f1* :: *F1*

```
  and   s1 :: nat
assumes l1-input-notempty-def: nat1 s1
  and   l1-invariant-def    : F1-inv f1
```

The individual operations are then specified as localte extensions and the post-conditions are given as definitions within the locale:

```
locale level1-new = level1-basic +
  assumes l1-new1-precondition-def: new1-pre f1 s1

locale level1-dispose = level1-basic +
   fixes d1 :: Loc
  assumes l1-dispose1-precondition-def: dispose1-pre f1 d1 s1

definition (in level1-new)
  new1-postcondition :: F1 ⇒ nat ⇒ bool
where
  new1-postcondition f' r ≡ new1-post f1 s1 f' r ∧ F1-inv f'

definition (in level1-dispose)
  dispose1-postcondition :: F1 ⇒ bool
where
  dispose1-postcondition f' ≡ dispose1-post f1 d1 s1 f' ∧ F1-inv f'

definition (in level1-dispose)
  dispose1-postconditionpsg :: F1 ⇒ bool
where
  dispose1-postconditionpsg f' ≡ dispose1-post2 f1 d1 s1 f' ∧ F1-inv f'
```

As in level 0, the feasibility proof operations are encoded as definitions as follows:

```
definition (in level1-new)
  PO-new1-feasibility :: bool
where
  PO-new1-feasibility ≡ (∃· f' r' . new1-postcondition f' r')

definition (in level1-dispose)
  PO-dispose1-feasibility :: bool
where
  PO-dispose1-feasibility ≡ (∃· f' . dispose1-postcondition f')

definition (in level1-dispose)
  PO-dispose1-feasibilitypsg :: bool
where
  PO-dispose1-feasibilitypsg ≡ (∃· f' . dispose1-postconditionpsg f')
```

5.3.3 Summary

The translation from VDM to Isabelle is relatively straightforward and faithful to the original model. Operations in VDM have a fairly natural translation to Isabelle's locale module system, where definitions can be used for the post-condition. It is future work to build a VDM package on top of Isabelle that would enable a syntactic emulation of VDM operations, thus reducing the chance of a human error in the translation (we, for example, forgot the invariant on our first iteration). While our strategy of packaging up preconditions, postconditions, and the invariants in definitions makes for additional proof steps, it ensures a comparmentalised proof and constructs explicit 'zoom' levels to have a clear domain of discourse. Additionally, our naming scheme makes it relatively straightforward to pick a definition 'from the air' and have

it be the right one, an oft overlooked but crucial requirement when models become large. The next section details the Isabelle proofs of the proof obligations for the above model, including:

- Feasibility proofs for both operations for both levels;
- Adaquecy proof for the reification;
- Widen-precondition for both operations;
- Narrow-postcondition for both operations;
- Sanity proofs that state that, for example, $DISPOSE(NEW) = Id$.

5.4 Proof of some properties of interest

In this section we prove some properties about the state invariant and operations that should hold. These kind of properties are problem specific and are useful to test the usefulness of the model (i.e. it's pragmatics). They are quite important, since we could prove something useless that is feasible and sound[6]!

5.4.1 Invariant testing

First, we test the Isabelle maps are good enough for our need to represent VDM maps in Isabelle. It would be useful to use the Isabelle value feature wrapping values with predicates like the invariant or the post condition.

Unfortunately, they are not enumerable (? TODO: Or just code not proved yet?). Instead, we prove that the invariant holds (and fails to hold) for certain values. This performs both positive and negative testing on the invariant. Proofs are automatic by auto.

value $[0 \mapsto 4,\ 6 \mapsto 11]$

definition
  F1-ex :: *F1*
where
  $F1\text{-}ex \equiv [0 \mapsto 4,\ 6 \mapsto 11]$

definition
  F1-ex-inv :: $F1 \Rightarrow bool$
where
  F1-ex-inv f $\equiv$ *F1-inv f*

lemmas *F1-ex-inv-defs = F1-ex-inv-def F1-inv-defs F1-ex-def*

5.4.2 Operations properties

Next, we prove some useful properties that operators at level 1 must satisfy. Incidentally, the proof of these properties helped hightlight various (general) lemmas about VDM maps missing in Isabelle.

[6]This has actually happened in a first version of the (wrong) model. That is we build the model right, but we didn't build the right model!

5.4.2.1 NEW 1 shrinks the memory

Upon memory allocation the resulting available memory **must** shrink. At first we tried something hard that often happens during proof: to prove a non-theorem (!) That is, to show that $f1' \subseteq_m f1$, which is of course false for the greater case. Nonetheless, this was useful to identify key missing lemmas for VDM maps, which were added to our library in theory VDMMaps.

In normal practice, it's important to use nitpick and quickcheck to try and invalidate our theorem by finding counter examples: these tools are much better at spotting non-theorems (with complicated assumptions) than normal users.

Our current version states that the resulting map must be different from the original (i.e. the allocation operation does something), and that its result leads to a subset of available locations ($locs\ f1' \subset locs\ f1$). Incidentally, *locs f1* is the retrieve function between level 0 and 1.

Proving proper subset is divided in two cases as subset and not equal. In these proofs, we decided to follow some advice given by Alan Bundy: "it is often useful [for learning/generalising] to have more than one proof for the same goal". We decided to take his suggestion and produce such variety, and in a truly novel form rather than just an artificial "reproving". Leo proved these goals as: i) "head-on", i.e. expanding and simplifying as we went; ii) "planned", i.e having an idea of what we wanted to achieve at each step and convincing Isabelle (often with extra lemmas) along the way; iii) "algebraically", i.e. having lemmas that chisel away operators to achieve what Alan calls "get rid of difficult operators"[7].

Moreover, independently, Iain is doing proofs by trying to "explain" the proof through Isar's declarative features to unpick the problem in yet another format. We also set it as a task for an MSc student that was not exposed to proof before (i.e. what we could expect of a well educated and motivated engineer): she (Nataliia) is doing them on her own after discussion and advice from Leo. The result [Sle13] is a pedagogical explanation of the proof process in line with Naur's [Nau72] from the perspective of a non-expert, well trained engineer. This last interaction could be taken us an expert training an engineer to handle/tackle proof and collecting the effort. Both Nataliia and Leo are running the proofs through Andrius' Isabelle/Eclipse-PP[8] [Vel12, Vel14], which captures the proof process by having a history log and encoding of attempts and features according to our MWhy models [JFV13].

Next, our aim is to study this data and try to infer general patterns from both PP data for comparison and fine tuning for learning techniques to take over [Gro12, GKL13, HK13]

context *level1-new*
begin

definition
PO-new1-postcondition-state-changes :: *nat* ⇒ *bool*
where
PO-new1-postcondition-state-changes r ≡ (∀· *f1'* . *new1-postcondition f1' r* ⟶ *f1'* ≠ *f1*)

definition
PO-new1-postcondition-state-locs-subset :: *nat* ⇒ *bool*
where
PO-new1-postcondition-state-locs-subset r ≡ (∀· *f1'* . *new1-postcondition f1' r* ⟶ *locs f1'* ⊆ *locs f1*)

definition
PO-new1-postcondition-diff-f-locs :: *nat* ⇒ *bool*
where

[7] This is a reference to trick by mathematicians trying to avoid complex operators. For instance, instead of proving the square root (e.g $\sqrt{2} = x$) of something they get rid of the square root by squaring both sides (e.g. $2{=}x^2$).

[8] Our Eclipse-based proof process (PP) collection environment that wraps around Isabelle's kernel for "tapping the wire" for information. It can be downloaded at `https://github.com/andriusvelykis/proofprocess`.

PO-new1-postcondition-diff-f-locs $r \equiv (\forall \cdot f1'$. *new1-postcondition* $f1'\ r \longrightarrow$ *locs* $f1' \neq$ *locs* $f1)$

definition
PO-new1-postcondition-shrinks-f-locs :: *nat* ⇒ *bool*
where
PO-new1-postcondition-shrinks-f-locs $r \equiv (\forall \cdot f1'$. *new1-postcondition* $f1'\ r \longrightarrow$ *locs* $f1' \subset$ *locs* $f1)$

definition
PO-new1-postcondition-f-equiv :: *nat* ⇒ *bool*
where
PO-new1-postcondition-f-equiv $r \equiv (\forall \cdot f1'$. *new1-postcondition* $f1'\ r \wedge$ *the*$(f1\ r) = s1 \longrightarrow \{r\}$ -◁ $f1' = \{r\}$ -◁ $f1)$

end

definition
PO-new1-dispose1-identity-post :: *F1* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where
PO-new1-dispose1-identity-post $f\ n\ r \equiv (\forall \cdot f'\ f''$. *new1-post* $f\ n\ f'\ r \wedge$ *dispose1-post* $f'\ r\ n\ f'' \wedge$ *F1-inv* $f \wedge$ *nat1* $n \longrightarrow f = f'')$

definition
PO-new1-dispose1-identity-pre :: *F1* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where
PO-new1-dispose1-identity-pre $f\ n\ r \equiv (\forall \cdot f'$. *new1-pre* $f\ r \wedge$ *new1-post* $f\ n\ f'\ r \wedge$ *F1-inv* $f \wedge$ *nat1* $n \longrightarrow$ *dispose1-pre* $f\ r\ n)$

Chapter 6

Heap proofs in Isabelle

6.1 Introduction

In this chapter, we describe the proof obligations and their proofs in Isabelle. For each of the main proof obligations, we give a high-level overview of the proof in terms of informal proof strategies, including the 'expert' motivations behind each proof step, corresponding to strategies and 'whys' in Chapter 3.

6.2 Feasibility proofs

There are four feasibility proofs: one for each operation of each level. Level 0 POs are trivial since there is no state invariant: they involve basic set theory. Isabelle can (almost) automatically discharge them. We just need to guide the necessary definition unfoldings. Level 1 POs, on the other hand, are more interesting and we concentrate on them below.

6.2.1 NEW 1 feasibility

The feasibility PO for the NEW operation states that (when all definitions have been unpacked):

$$
\begin{array}{l}
\forall \cdot f\ s.\ \mathit{F1\text{-}inv}\ f \wedge \mathit{nat1}\ s \wedge (\exists \cdot l \in \mathit{dom}\ f.\ s \leq \mathit{the}\ (f\ l)) \longrightarrow \\
\quad (\exists \cdot f'\ r. \\
\qquad r \in \mathit{dom}\ f \wedge \\
\qquad (\mathit{the}\ (f\ r) = s \wedge f' = \{r\} \lhd\!\text{-}\ f \vee \\
\qquad\ s < \mathit{the}\ (f\ r) \wedge f' = \{r\} \lhd\!\text{-}\ f \cup m\ [r + s \mapsto \mathit{the}\ (f\ r) - s]) \wedge \\
\qquad \mathit{F1\text{-}inv}\ f')
\end{array}
$$

This is not dissimilar to expanding definitions from the general PO form given in Appendix A. The first thing to note is that the conclusion contains a disjunction and can be rewritten to:

$$
\begin{array}{l}
\forall \cdot f\ s.\ \mathit{F1\text{-}inv}\ f \wedge \mathit{nat1}\ s \wedge (\exists \cdot l \in \mathit{dom}\ f.\ s \leq \mathit{the}\ (f\ l)) \longrightarrow \\
\quad (\exists \cdot f'\ r.\ r \in \mathit{dom}\ f \wedge \mathit{the}\ (f\ r) = s \wedge f' = \{r\} \lhd\!\text{-}\ f \wedge \mathit{F1\text{-}inv}\ f') \vee \\
\quad (\exists \cdot f'\ r. \\
\qquad r \in \mathit{dom}\ f \wedge \\
\qquad s < \mathit{the}\ (f\ r) \wedge \\
\qquad f' = \{r\} \lhd\!\text{-}\ f \cup m\ [r + s \mapsto \mathit{the}\ (f\ r) - s] \wedge \mathit{F1\text{-}inv}\ f')
\end{array}
$$

This can be seen as a semantics preserving transformation on the feasibility goal. It can be proved as an identity to be applied. The reason (why) for performing this transformation, which we could call 'distribute existentials over disjunctions' is because it is possible that each

part of the disjunction would need a slightly different witness. In fact, in this case, it is pretty obvious that we might want to do this, since there are explicit single-point instantiations for the existential on each part of the disjunction: $f' = \{r\} \lhd f$ and $f' = \{r\} \lhd f \cup m\ [r + s \mapsto \mathit{the}\ (f\ r) - s]$. In general, this is not the case and the user may be required to provide a more subtle (non-deterministic) witness.

In a larger example what usually happens is that some variables are one-point-ruled away, hence constraining remaining existentialy quantified variables values to be given by the user in an explicit existential introduction step. Worse, depending on the layers of definitions used, the disjunction might not be obvious. For instance, top-level feasibility POs in the Mondex case study [FW08] have over 1200 existentially quantified variables too many predicates to count, of which 729 explicit instantiation need to be provided by the user, if done naively. Careful consideration and attention to various layers of interest was crucial to cope with the goal complexity. Identifying such proof intent ("why" meta-data) would guide our tools in the search for similar proof strategies for such goals.

Just choosing one side of the disjunction is going to lead us into difficulty, because of the $s < \mathit{the}\ (f\ r)$ or $\mathit{the}\ (f\ r) = s$ part of the goals. In the assumptions we have only $s \leq \mathit{the}\ (f\ r)$. This suggests a hidden case analysis on the $\leq$, leading to the revised goal (which is then split into two subgoals using disjunction elimination):

$$
\begin{array}{l}
\forall \cdot f\ s.\ \mathit{F1\text{-}inv}\ f \wedge \mathit{nat1}\ s \wedge (\exists \cdot l \in \mathit{dom}\ f.\ \mathit{the}\ (f\ l) = s \vee s < \mathit{the}\ (f\ l)) \longrightarrow \\
\quad (\exists \cdot f'\ r.\ r \in \mathit{dom}\ f \wedge \mathit{the}\ (f\ r) = s \wedge f' = \{r\} \lhd f \wedge \mathit{F1\text{-}inv}\ f') \vee \\
\quad (\exists \cdot f'\ r. \\
\qquad r \in \mathit{dom}\ f \wedge \\
\qquad s < \mathit{the}\ (f\ r) \wedge \\
\qquad f' = \{r\} \lhd f \cup m\ [r + s \mapsto \mathit{the}\ (f\ r) - s] \wedge \mathit{F1\text{-}inv}\ f')
\end{array}
$$

which give us a natural choice of disjunct for introduction in each goal.

We use the term "hidden case distinction" (another 'why') here, because there is no explicit disjunction in the assumptions. Rather, we apply a lemma which states:

$$(y \leq x) = (x = y \vee y < x)$$

to make it clear. In general, we may need to apply some additional transformations or deeper analysis to make clear the disjunction. Or, it may require a complicated theorem. In this case, we simply need to apply the intro tactic to deal with the universal quantifiers, implication, and conjunctions to expose the new disjunction. The final step of the hidden case analysis is to apply disjunction elimination. In the DISPOSE operation there are two hidden case distinctions. We discuss this (reused) strategy further there. We now have two subgoals:

1. The first goal we use disjunction introduction and choose to solve the equals case, instantiating r as the l in the assumptions and f' as the appropriate one point witness i.e. $f' = \{r\} \lhd f$ allows us to discharge the first two conjuncts of the goal trivially. The third — the invariant — is basically $\mathit{F1\text{-}inv}\ (\{r\} \lhd f)$, which unfolds as:

   $$
   \begin{array}{l}
   \mathit{Disjoint}\ (\{r\} \lhd f) \wedge \\
   \mathit{sep}\ (\{r\} \lhd f) \wedge \mathit{nat1\text{-}map}\ (\{r\} \lhd f) \wedge \mathit{finite}\ (\mathit{dom}\ (\{r\} \lhd f))
   \end{array}
   $$

   under the assumption $\mathit{Disjoint}\ f \wedge \mathit{sep}\ f \wedge \mathit{nat1\text{-}map}\ f \wedge \mathit{finite}\ (\mathit{dom}\ f)$.

   Attempting to solve one of these suggests the general structure of lemmas to solve them all:

   $$\mathit{Disjoint}\ f \Longrightarrow \mathit{Disjoint}\ (s \lhd f)$$

where s is a set of locations. The idea here is a strategy called invariant breakdown [1] which conjectures lemmas about the invariant over the map operators. The idea being that it can be eventually broken down to the extent where the assumption about the invariant on the original domain will hold. The 'why' for using this strategy is when the updated state is constructed from modifications to the original (map operators in our case). This, of course, need not necessarily be the case, but turns out to be true for all the operations in this case study, and is often the case in other larger examples [FW08, WF08, BFW09, FW09].

Because we encoded the individual parts of our invariant as definitions, we can apply this strategy in a modular fashion for each of the four invariant parts. That is a key reason "why" having zoom levels is useful: the updated invariant without the zoom-layers of definitions would look like this:

$$
\begin{array}{l}
(\forall\, a{\in}\mathit{dom}\ (\{r\} \lhd\!\!- f).\\
\quad \forall\, b{\in}\mathit{dom}\ (\{r\} \lhd\!\!- f).\\
\qquad a \neq b \longrightarrow \mathit{disjoint}\ (\mathit{Locs\text{-}of}\ (\{r\} \lhd\!\!- f)\ a)\ (\mathit{Locs\text{-}of}\ (\{r\} \lhd\!\!- f)\ b)) \wedge\\
(\forall\, l{\in}\mathit{dom}\ (\{r\} \lhd\!\!- f).\ l + \mathit{the}\ ((\{r\} \lhd\!\!- f)\ l) \notin \mathit{dom}\ (\{r\} \lhd\!\!- f)) \wedge\\
(\forall\, x.\ x \in \mathit{dom}\ (\{r\} \lhd\!\!- f) \longrightarrow \mathit{nat1}\ (\mathit{the}\ ((\{r\} \lhd\!\!- f)\ x))) \wedge\\
\mathit{finite}\ (\mathit{dom}\ (\{r\} \lhd\!\!- f))
\end{array}
$$

In a more complicated situation like the Mondex example, a naive full exapansion of the predicate goal needs GB of memory loads of CPU time and 45 pages of A4! Creating this layers in examples like this is vital. Here, it keeps proof repetition and drudgery to a minimum. It also aids our (still under development) strategy matching algorithms with new goals given previously known/declared "why"s.

The proofs of the lemmas for $\mathit{nat1\text{-}map}\ (\{r\} \lhd\!\!- f)$ and $\mathit{finite}\ (\mathit{dom}\ (\{r\} \lhd\!\!- f))$ are trivial; the other two are more complicated, but can still be solved by Isabelle's automation and do not require any additional side conditions. In the development, these are represented as four lemmas

$$\mathit{nat1\text{-}map}\ f \Longrightarrow \mathit{nat1\text{-}map}\ (s \lhd\!\!- f)$$

$$\mathit{finite}\ (\mathit{dom}\ f) \Longrightarrow \mathit{finite}\ (\mathit{dom}\ (s \lhd\!\!- f))$$

$$\mathit{Disjoint}\ f \Longrightarrow \mathit{Disjoint}\ (s \lhd\!\!- f)$$

and $\mathit{sep}\ f \Longrightarrow \mathit{sep}\ (s \lhd\!\!- f)$.

For the map anti-restriction operation, we only require the $P\ f$ assumption to show $P\ (s \lhd\!\!- f)$; in general, subtle side-conditions may be required, which is where the work of this proof really lies. Finally, we mention that Isabelle can prove these four lemmas automatically beased on the VDM Maps library that we have provided. More realistically, at first iteration, these goals served to shape what kind of general map lemmas we needed!

2. For the second goal, we again use invariant breakdown. In this case, however, the updated state is more complicated. As a result the invariant conditions are more complicated:

$$\mathit{Disjoint}\ (\{r\} \lhd\!\!- f \cup m\ [r + s \mapsto \mathit{the}\ (f\ r) \mathbin{-} s])$$

Again, in this case, a single lemma suggests the approach for all the rest, under the assumption $\mathit{Disjoint}\ f$:

[1] Could also be seen as a poor mans rippling

$Disjoint\ (f \cup m\ [a \mapsto b])$

Now, this lemma is suggested in analogy with the previous sub-goal case. We can prove that the assumption holds using the lemma from the first goal and our assumption. To prove this lemma, we need extra conditions, however:

$a \notin dom\ f$

$nat1\text{-}map\ f$

$nat1\ b$

$disjoint\ (locs\text{-}of\ a\ b)\ (locs\ f)$

The first comes from the side condition that map union domains must be disjoint. The second and third comes from the definition of *Disjoint*, which involves *locs-of* $(x\ (the\ (f\ x)))$ that requires the map is on $\mathbb{N}_1$ range and the second argument being greater than zero. The final condition relates to the precondition of dispose, which is required in order to make the state update under the invariant possible.

To show that these hold in the current proof obligation is relatively straightforward and each can be solved by Isabelle's automation. To prove the lemma itself, on the other hand, is not so straightforward. It needs case analysis and some detailed reasoning.

The *sep* part of the invariant is similar to Disjoint and needs an analagous lemma albeit with different conditions, which are likewise mostly solved by Isabelle's automation. Another part of the AI4FM project dealing with implicit strategies hopes to develop techniques for learning analagous lemmas; we hope that we can utilise this approach to suggest side-conditions. The invariant breakdown strategy provides a clear route through this proof. Now, most of the work by an 'expert' is in conjecturing the right conditions for the lemmas, as well as any needed (VDM map) datatype general lemmas. An alternative approach, though naive and cumbersome, would be to include all global assumptions in the suggested lemma. Once the lemma has been proved (if it is valid) one can analyse for unused assumptions. Such a transformation has been suggested by Whiteside as a proof refactoring [Whi13]. In this case study, we attempted to gain an understanding of 'why' the lemma was true to arrive at a natural set of assumptions (especially as we envisage it may be reused). Another important consideration in the specification of lemma conditions involves the 'zoom-level' of the assumptions. For example, a lemma can be specified as[2]:

$VDM\text{-}F1\text{-}inv\ f \Longrightarrow P\ (\{r\} \lhd\!\text{-}\ f)$

or

$[\![sep\ f;\ Disjoint\ f]\!] \Longrightarrow P\ (\{r\} \lhd\!\text{-}\ f)$

which are equivalent, but the unfolding of *VDM-F1-inv* must occur at the top-level or in the proof of the lemma; similarly, we could decide to weaken the lemma by passing a strong assumption (the full *F1-inv* for example) if we always expect it to be used in a context where the invariant holds.

[2]Isabelle represents chains of assumptions using $[\![A;B;C]\!] \Longrightarrow D$ to mean $A, B, C \vdash D$

6.2.2 DISPOSE 1 feasibility

Far more complicated in appearance, but only requiring one new idea is the DISPOSE feasibility proof. The PO is as follows:

$$
\begin{array}{l}
\forall \cdot f\ d\ s. \\
\quad \mathit{F1\text{-}inv}\ f \wedge \mathit{nat1}\ s \wedge \mathit{disjoint}\ (\mathit{locs\text{-}of}\ d\ s)\ (\mathit{locs}\ f) \longrightarrow \\
\quad (\exists \cdot f'.\ f' = \\
\qquad (\mathit{dom}\ (\mathit{dispose1\text{-}below}\ f\ d) \cup \mathit{dom}\ (\mathit{dispose1\text{-}above}\ f\ d\ s))\ \text{-}\!\triangleleft\ f \cup m \\
\qquad [\mathit{min\text{-}loc}\ (\mathit{dispose1\text{-}ext}\ f\ d\ s) \mapsto \\
\qquad\ \mathit{HEAP1.sum\text{-}size}\ (\mathit{dispose1\text{-}ext}\ f\ d\ s)] \wedge \\
\qquad \mathit{F1\text{-}inv}\ f')
\end{array}
$$

which, when the appropriate introduction rules and the one-point existential witness is supplied, is basically the following goal:

$$
\begin{array}{l}
\mathit{F1\text{-}inv} \\
((\mathit{dom}\ (\mathit{dispose1\text{-}below}\ f\ d) \cup \mathit{dom}\ (\mathit{dispose1\text{-}above}\ f\ d\ s))\ \text{-}\!\triangleleft\ f \cup m \\
\ [\mathit{min\text{-}loc}\ (\mathit{dispose1\text{-}ext}\ f\ d\ s) \mapsto \mathit{HEAP1.sum\text{-}size}\ (\mathit{dispose1\text{-}ext}\ f\ d\ s)])
\end{array}
$$

It is actually of the same shape as the second case for NEW feasibility (an anti-restricted map extended with a singleton set). Pause to think how would this goal look like without the folded definitions for *above* and *below*:

$$
\begin{array}{l}
\mathit{F1\text{-}inv} \\
((\mathit{dom}\ (\{x \in \mathit{dom}\ f \mid x + \mathit{the}\ (f\ x) = d\} \triangleleft f) \cup \\
\quad \mathit{dom}\ (\{x \in \mathit{dom}\ f \mid x = d + s\} \triangleleft f))\ \text{-}\!\triangleleft \\
\ f \cup m \\
\ [\mathit{min\text{-}loc} \\
\quad (\{x \in \mathit{dom}\ f \mid x + \mathit{the}\ (f\ x) = d\} \triangleleft f \cup m\ \{x \in \mathit{dom}\ f \mid x = d + s\} \triangleleft f \cup m \\
\qquad [d \mapsto s]) \\
\ \mapsto \mathit{HEAP1.sum\text{-}size} \\
\qquad (\{x \in \mathit{dom}\ f \mid x + \mathit{the}\ (f\ x) = d\} \triangleleft f \cup m\ \{x \in \mathit{dom}\ f \mid x = d + s\} \triangleleft f \cup m \\
\qquad\ [d \mapsto s])])
\end{array}
$$

It is clearly more difficult to spot such similarities with $NEW1$ without the zoom layers around key concepts in formulae. Moreover, if we (naively) throw Isabelle's heaviest tool (auto) at the goal, we would get 4 subgoals fitting a two page of A4!

Thus, the same invariant breakdown strategy *could* be used here, using the lemmas that the expert conjectured for the $NEW1$ feasibility proof. However, we do not apply this strategy just yet. The reason behind this is that there are two hidden case distinctions that significantly simplify the proof obligations. These are on the shape of *dispose1-below f d* and *dispose1-above f d s*. Recall the definitions:

$$\mathit{dispose1\text{-}below}\ f\ d \equiv \{x \in \mathit{dom}\ f \mid x + \mathit{the}\ (f\ x) = d\} \triangleleft f$$

$$\mathit{dispose1\text{-}above}\ f\ d\ s \equiv \{x \in \mathit{dom}\ f \mid x = d + s\} \triangleleft f$$

The filtering equalities force *above* and *below* to either be empty or a singleton set. Thus, the top level strategy here is to perform case analysis on these maps. For the case that both are empty, things simplify out nicely (*e.g.* the anti restriction

$$(\mathit{dom}\ (\mathit{dispose1\text{-}below}\ f\ d) \cup \mathit{dom}\ (\mathit{dispose1\text{-}above}\ f\ d\ s))\ \text{-}\!\triangleleft\ f$$

disappears because domain of empty is empty and subtracing empty is unit law for antirestriction).

We describe the technique for solving the case where $\mathit{dispose1\text{-}below}\ f\ d = \{\}$ and

$dispose1\text{-}above\ f\ d\ s \neq \{\}$

. From the definition of *dispose1-above* we know that it is a singleton with domain $\{d{+}s\}$. This also allows us to reason about *min-loc* (*dispose1-ext f d s*) and *HEAP1.sum-size* (*dispose1-ext f d s*). Recall the definition of *dispose1-ext*:

$dispose1\text{-}ext\ f\ d\ s \equiv dispose1\text{-}above\ f\ d\ s \cup m\ dispose1\text{-}below\ f\ d \cup m\ [d \mapsto s]$

This means that we also know that the *min-loc* (*dispose1-ext f d s*) = *d*. We also know that *HEAP1.sum-size* (*dispose1-ext f d s*) = $s + the\ (f\ (d + s))$. Putting this information together, we get the proof obligation (for *sep*) as:

$sep\ (\{d1 + s1\}\ \text{-}\!\lhd\ f1 \cup m\ [d1 \mapsto the\ (f1\ (d1 + s1)) + s1])$

which is considerably simpler. In order to expose this as the true proof obligation (under the case analysis), a strategy called we call *shaping* (or directed substitution) is used. In a shaping strategy, subterms of the goal are proved to be equal to expert-supplied terms and substituted in to form the new (simpler) goal, under (locale) specific assumptions. In this case there are three shaping lemmas:

$dom\ (dispose1\text{-}below\ f\ d) \cup dom\ (dispose1\text{-}above\ f\ d\ s) = \{d + s\}$

$min\text{-}loc\ (dispose1\text{-}ext\ f\ d\ s) = d$

and

$HEAP1.sum\text{-}size\ (dispose1\text{-}ext\ f\ d\ s) = s + the\ (f\ (d + s))$

The same techniques apply to the other cases to get slightly different 'shaped' lemmas. At this point, with the shaped PO, we can begin the invariant breakdown strategy. As before, the *nat1* and *finite* parts of the invariant are trivial. The difficulty is with *sep* and *Disjoint*.

For example, the side-conditions for

$[\![a \notin dom\ f;\ sep\ f;\ \forall\, l{\in}dom\ f.\ l + the\ (f\ l) \notin dom\ [a \mapsto b];\ a + b \notin dom\ f;$
$nat1\ b]\!]$
$\Longrightarrow sep\ (f \cup m\ [a \mapsto b])$

are:

1. $d1 \notin dom\ (\{d1 + s1\}\ \text{-}\!\lhd\ f1)$, which is easy to solve by automation.
2. $sep\ (\{d1 + s1\}\ \text{-}\!\lhd\ f1)$ is solved by further application of invariant breakdown using the before-state invariant hypothesis (*F1-inv f*).
3. $nat1\ (the\ (f1\ (d1 + s1)) + s1)$, which is straightforward for automation to solve.
4. $\forall\, l{\in}dom\ (\{d1 + s1\}\ \text{-}\!\lhd\ f1).\ l + the\ ((\{d1 + s1\}\ \text{-}\!\lhd\ f1)\ l) \notin dom\ [d1 \mapsto the\ (f1\ (d1 + s1)) + s1]$, which requires some work.
5. $d1 + (the\ (f1\ (d1 + s1)) + s1) \notin dom\ (\{d1 + s1\}\ \text{-}\!\lhd\ f1)$, which also requires effort.

The last two generated subgoals correspond to showing that a) there is no chunk of memory in the free store that touches the start (domain) of the singleton to be added; and, b) the last element in the singleton does not touch any start locations in the free store (i.e. the following element must not be in the domain). That is to say, adding this new element to the map really does keep it separate: it does not touch anything on either end. It is at this point that Freitas introduced a new concept called *sep0* that gave a uniform definition for reasoning about this concept. In Whiteside's development, however, these subgoals were solved manually.

In retrospect, a definition to refer to the loction straight after a chunk of memory may have clarified these conditions e.g. *after s f* = *s* + *the* (*f s*) would simplify the first tricky condition to:

$\forall \cdot l \in dom\ (\{d1 + s1\} \mathbin{-\!\!\triangleleft} f1).\ after\ l\ (\{d1 + s1\} \mathbin{-\!\!\triangleleft} f1) \notin dom\ [d1 \mapsto the\ (f1\ (d1 + s1)) + s1]$

For this condition, the goal comes down to showing that for any $l \in dom\ f$ we have $l + the\ (f1\ l) \neq d1$[3]. This is because we can rewrite membership of a singleton domain as an equality and because if $l + the\ (f1\ l) \neq d1$ then $l + the\ ((\{d1 + s1\} \mathbin{-\!\!\triangleleft} f1)\ l) \neq d1$ and we assume that $l \neq d1 + s1$. Now, since we are under the assumption that *dispose1-below f d* = $\{\}$ and since *dispose1-below f d* $\equiv \{x \in dom\ f \mid x + the\ (f\ x) = d\} \triangleleft f$, the result follows easily.

For the final goal, the sep part of the invariant allows us to conclude that $d1 + s1 + the\ (f1\ (d1 + s1)) \notin dom\ f1$, which implies that $d1 + s1 + the\ (f1\ (d1 + s1)) \notin dom\ (\{d1 + s1\} \mathbin{-\!\!\triangleleft} f1)$ since the antirestricted domain is a subset of the full domain; we can conclude by simple associative/commutative-rewriting with plus. For the other cases, where *dispose1-above f d s* $\neq \{\}$ etc, we follow exactly the same strategies, with minor differences, but with loads of drudgery (e.g. about 3/4 of the proof script) and fewer, if any, new ideas needed.

6.3 Level 0 and level 1 reification

The next set of proof obligations are the reification proof obligations between levels 0 and 1. There are three types of proof obligation:

- Adaquecy: shows that there is a level 1 state to match every level 0 state (and such that the invariant holds). Because the retrieve is a function, it also means such chosen link between types in this case is unique.
- Widen-precondition: concrete assumptions must be the same as or weaker than abstract assumptions.
- Narrow-postcondition: concrete commitments must be the same as or stronger than abstract commitments.

They justify the change in datatype representation by keeping models between levels compatible.

6.3.1 Adequacy

The proof obligation is $\exists \cdot !f1.\ f0 = retr0\ f1 \wedge F1\text{-}inv\ f1$ (where the uniqueness isn't required, but we have it anyway as we can prove it). The goal states that the retrieve function linking the two state representations is unique and satisfy the concrete invariant. The top level strategy for this proof is a custom induction rule applied to *f0* that operates on finite, contiguous, non-abutting sets. The rule looks like

$$\begin{array}{l}[\![finite\ F;\ P\ \{\};\\ \bigwedge F\ F'.\\ \quad [\![finite\ F;\ finite\ F';\ F' \neq \{\};\ contiguous\ F';\ non\text{-}abut\ F\ F';\ P\ F]\!]\\ \quad \Longrightarrow P\ (F \cup F')]\!]\\ \Longrightarrow P\ F\end{array}$$

and is provided and proved by the expert. Then, the empty case is simple to prove: the required witness for *f1* is the empty map. For the step case, we need to show, under the induction hypotheses:

F = *retr0 f1hook*

[3] Or *after l f1* $\neq$ *d1*.

F1-inv f1hook

contiguous F′

non-abut F F′

that

$\exists\cdot!f1.\ F \cup F' = retr0\ f1 \wedge F1\text{-}inv\ f1$

The key observation is to apply the witnessing strategy with the appropriate value of a witness. In this case, we do not have a one-point rule that makes it clear. Instead, the expert has to provide it:

$f1 = f1hook \cup m\ [Min\ F' \mapsto card\ F']$

As a justification for pulling this witness out of the air, recall the definition of the retrieve function:

$retr0\ f1 = locs\ f1$

and note that it is a reasonable 'intuition', perhaps, that this conjecture is true:

$locs\ (f \cup m\ g) = locs\ f \cup locs\ g$

therefore we just need to show that:

$F = locs\ f1hook$

and

$F' = locs\ [Min\ F' \mapsto card\ F']$

The first is precisely the induction hypothesis. For the second subgoal, we conjecture that $F' = locs\text{-}of\ (Min\ F')\ (card\ F')$, which is intuitively true. Recall that the cardinality of a set is the number of elements and the *Min* function in Isabelle returns the minimum element of a finite set. Thus, $locs\text{-}of\ (Min\ F')\ (card\ F')$ gives us a contiguous set of length $card\ F'$ starting from $Min\ F'$.

Recall the induction assumption states that F' is contiguous (as defined by $contiguous\ ?F \equiv \exists\cdot m\ l.\ nat1\ l \wedge ?F = locs\text{-}of\ m\ l$), and allows us to solve the goal (since the *locs* of a singleton is simply *locs-of*). This leaves us with two lemmas to prove (with possible side-conditions):

1. $locs\ (f \cup m\ g) = locs\ f \cup locs\ g$. Actually, we proved a more specific lemma:

   $locs\ (f \cup m\ [x \mapsto y]) = locs\ f \cup locs\text{-}of\ x\ y$

   which just requires the assumption

   $x \notin dom\ f$

   to ensure the map union is well-formed. The proof of this lemma is a straightforward piece of algebraic reasoning. Unfolding the definition of locs, we get a union of all *locs-of* over the domain of the map:

   $locs\ (f \cup m\ [x \mapsto y]) =$
   $(\bigcup_{s \in dom\ (f \cup m\ [x \mapsto y])}\ locs\text{-}of\ s\ (the\ ((f \cup m\ [x \mapsto y])\ s)))$

Now, we can easily show that the $dom\ (f \cup m\ [x \mapsto y]) = \{x\} \cup dom\ f$, and then that:

$(\bigcup_{s \in \{x\} \cup dom\ f}\ locs\text{-}of\ s\ (the\ ((f \cup m\ [x \mapsto y])\ s))) =$
$locs\text{-}of\ x\ (the\ ((f \cup m\ [x \mapsto y])\ x)) \cup$
$(\bigcup_{s \in dom\ f}\ locs\text{-}of\ s\ (the\ ((f \cup m\ [x \mapsto y])\ s)))$

where the second union is simply $locs\ f$ and we are done.

2. $contiguous\ F' \Longrightarrow locs\text{-}of\ (Min\ F')\ (card\ F') = F'$ is solved with the help of two lemmas: one showing that $Min\ (locs\text{-}of\ m\ l) = m$ and the other that $card\ (locs\text{-}of\ m\ l) = l$. Both these lemmas are proved by a simple induction on l.

Both these lemmas allow us to conclude the first part of the proof. The overall idea of this part of the proof was to translate the $\cup m$ operator to $\cup$ and show that both sides were equal in:

$F \cup F' = locs\ (f1hook \cup m\ [Min\ F' \mapsto card\ F'])$

The next step is then to show that the invariant holds. That is:

$F1\text{-}inv\ (f1hook \cup m\ [Min\ F' \mapsto card\ F'])$

To solve this goal, we break down the definition and solve each individual invariant part separately. We take $sep\ (f1hook \cup m\ [Min\ F' \mapsto card\ F'])$ as an example, and we follow the same invariant breakdown strategy as both the feasibility proof obligations (a map union extending a map with a singleton map). The two difficult side conditions for this invariant breakdown require effort. For example, one has to prove that:

$Min\ F' + card\ F' \notin dom\ f1hook$

We show this by a contradiction. Why do we try proof by contradiction here? Because of the $\notin$, certainly[4]. The contradiction constructed uses the abuttedness property of the induction rule:

$non\text{-}abut\ F\ F'$

where

$non\text{-}abut\ s1\ s2 \equiv$
$disjoint\ s1\ s2 \wedge (\forall \cdot l1 \in s1.\ \forall \cdot l2 \in s2.\ l1 + 1 < l2 \vee l2 + 1 < l1)$

First, we know that $l1 = Min\ F' + card\ F' - 1 \in F'$ and that $l2 = Min\ F' + card\ F' \in dom\ f1hook$ and that $dom\ f1hook \subseteq F$ therefore $Min\ F' + card\ F' \in F$. Now, by non-abuttedness, we know that $l1 + 1 < l2 \vee l2 + 1 < l1$, but this is a contradiction since $l1 + 1 = l2$.
To prove the (optional) uniqueness of the retrieve function, we use the theorem

$[\![locs\ f = locs\ g;\ F1\text{-}inv\ f;\ F1\text{-}inv\ g;\ f \neq Map.empty;\ g \neq Map.empty]\!] \Longrightarrow f = g$

which states that under the invariant equality of the locations implies equality of the maps.

[4]One might reasonably question why we didn't try proof by contradiction in the equivalent step in the feasibility POs?

6.3.2 Widen-precondition

For the NEW operations, the widen precondition proof requires us to show that if the NEW0 precondition holds then the NEW1 precondition holds:

PO-l01-new-widen-pre ≡
$\forall \cdot f1\ s1.\ F1\text{-}inv\ f1 \wedge nat1\ s1 \wedge new0\text{-}pre\ (retr0\ f1)\ s1 \longrightarrow new1\text{-}pre\ f1\ s1$

which unfolds to saying (under additional preconditions) if $\exists \cdot l.$ *is-block l s1* (*retro f1*) then $\exists \cdot l \in dom\ f1.\ s1 \leq the\ (f1\ l)$. That is, if there is a block in the set of locations defined by the retrieve function, then there is an element in the map that has a size large enough. It is tempting to assume that the l gained from existential elimination on the assumption is the one required as the witness in the conclusion, but this is not the case since there is no way to prove that $l \in dom\ f$. This was the first attempt at solving this proof and the incorrentess of the proof step showed itself immediately. Rather, the approach is more subtle: one has to maneuvere the goal to find the appropriate witness. The proof sketch used by Whiteside is as follows:

have *locs-subset*: *locs-of l s1* ⊆ *locs f1*
  sorry — Show that the locations are indeed with the free space
then have *l* ∈ *locs f1*
  sorry — Specifically, the first element is in it
then have *l* ∈ (⋃ *s*∈*dom f1*. *locs-of s* (*the* (*f1 s*)))
  sorry — Unfold the definition of locs
then have ∃· *m*∈ *dom f1*. *locs-of l s1* ⊆ *locs-of m* (*the* (*f1 m*))
  sorry — Show that *locs-of l s1* must be contained in **one** other *locs-of*
then obtain *m* **where** *mindom*: *m*∈*dom f1* **and**
      locssubm: *locs-of l s1* ⊆ *locs-of m* (*the* (*f1 m*))
  sorry — Then find an arbitrary m that contains the locations from l
then have *mgrs1*: *s1* ≤ *the* (*f1 m*)
  sorry — Show that s1 must be *s1* ≤ *m*

Note that the two facts *mindom* and *mgrs1* defined in the sketch[5] are exactly what is required to solve the goal. Most of the intermediate steps in this sketch are easily solved by automation. The final step requires extra work, as does showing:

$\exists \cdot m \in dom\ f1.\ locs\text{-}of\ l\ s1 \subseteq locs\text{-}of\ m\ (the\ (f1\ m))$

which is proved as a lemma that depends precisely on the invariant (requiring nested proof by contradictions). Before describing this final part of the proof, we consider the 'why' behind the above sketch. It is directly motivated by the original failed proof: we know that the locations are in the free store, and by the invariant, they must be within one other (possibly larger) set of locations. The bad assumption initially was simply that they were taken from the front of a set ($l \in dom\ f$). Thus, we really needed to find the domain element (*i.e.* witnessing), then show that its range is greater than or equal to $s1$.

To show this requires another hidden case analysis that is hinted at in the preconditions: either $l = m$ or $l < m$. In fact, our case analysis simply is on equals or not equals. For the equals case, we have a lemma

$[\![0 < x;\ 0 < y;\ locs\text{-}of\ l\ x \subseteq locs\text{-}of\ l\ y]\!] \Longrightarrow x \leq y$

and that does the job for us. For the not equals case, we first show that $m < l$ by contradiction, since if this is true then l would not be in the locations, but we have already shown that it is. After we have established this fact, we use another lemma

$[\![0 < x;\ 0 < y;\ l' < l;\ locs\text{-}of\ l\ x \subseteq locs\text{-}of\ l'\ y]\!] \Longrightarrow x \leq y$

[5] Recall the discussion about proof sketching in Section 5.2.3.1.

which is similar (analagous) to the previous case, and this completes the proof.

The dispose case is trivial since the preconditions are identical (once the retrieve function has been unfolded to *locs*.

6.3.3 Narrow postcondition

The narrow postcondition proof obligations state that if the post condition holds at level 1 then it will also hold at level 0 under the retrieve function. That is, for $NEW1$ is states that if *new1-post f1 s1 f1′ r* then

new0-post (*retr0 f1*) *s1* (*retr0 f1′*) *r*

We start by unfolding the $NEW1$ post condition, whcih is a disjunction (the equals case or the greater than case). This gives us an explicity case split. For each case we need to show the two parts of the $NEW0$ postcondition holds as:

is-block r s1 (*locs f1*)

and

locs f1′ = *locs f1* - *locs-of r s1*

where the *locs f1′* corresponds to the updated free store. The first subgoal is straightforward. For the second goal, we have the assumption that $f1' = \{r\} \lhd\!\!- f1$ and so we use the *dom-ar-locs* lemma to rewrite the $locs(f1')$ as:

⟦*finite* (*dom f*); *nat1-map f*; *Disjoint f*; *l* ∈ *dom f*⟧
⟹ *locs* ({*l*} -⊲ *f*) = *locs f* - *locs-of l* (*the* (*f l*))

The second case is more difficult but follows the same pattern:

$f1' = \{r\} \lhd\!\!- f1 \cup m\ [r + s1 \mapsto the\ (f1\ r) - s1]$

and we rewrite the locs of this using two lemmas. To complete the proof simply requires some algebraic manipulation and discharging the side-conditions, which is mostly automatable???.

For the dispose operation, the proof follows a similar technique. Instead of the case distinction on the postcondition (using an explicit disjunction) we have case analysis on *below* and *above* being the *Map.empty* map. Then, for each of the case we apply the same locs distribution lemmas as for new, perform algebraic manipulations, and discharge side-conditios. This is one of the longest proofs in the level 1, but requires the least thought!

tht properly discuss the strategies and reusability.

6.4 Summary

We have formalised level 0 and level 1 of the VDM model and their reification, including all of the generated proof obligations. We further satisfied ourselves of the validity of the model by proving various indentities that we expected to exist in the model: so-called 'sanity checks'.

Furthermore, we performed two parallel proof attempts. Freitas leveraged his experience in the Z method and the Z/Eves theorem prover [Saa97, Fre04], and pursued a traditional, procedural tactic-based style of proof[6]. Whiteside, who comes from the Isar school of Isabelle proof [Wen02], took a declarative, forwards approach to the proof obligations that was centered around proof sketches: high-level proof steps that solve the problem, but have *gaps* that must

[6] In fact, Freitas has also formalised the heap store in Z. See Appendix G and model evolution discussion in Chapter 4.

gradually be filled in. Our goal in pursuing parallel, stylistically different proof attempts was to understand more clearly how different experts would proceed and to gather additional data on the strategies employed. We wish to find if proof ideas (whys) transcend the details of a proof language and if particular patterns of proof have instantiations in different styles.

The result is an interesting story: broadly speaking, the proofs have the same *idea*, or *why*, but often diverge in some critical places. This divergance is mostly due to the proof language's style itself. In one proof, for example, Whiteside has a case distinction over the DISPOSE post-condition for proving the invariant holds on the updated state, resulting in an easy to understand proof; Freitas, on the other hand, introduces a specific lemma which crunches the case distinction by having complex side-conditions, losing understandability but shortening the proof considerably. In other cases it is the expert taking a different approach. For example, one 'expert' (Freitas) introduces a new concept that simplifies (and makes clearer) the *sep* part of the invariant proofs. In a final example, Whiteside uses expert knowledge of the proof situation to eliminate a complicated case distinction.

The Isabelle formalisation of the heap store also provided a compelling example of the need for formalisation, throwing up several issues with our original VDM model and requiring modifications to the model to be made. In several cases, the changes were trivial (a '+ 1' removed, for example); the NEW post-condition required a fairly substantial change. This chapter will not dwell on our failings, however, and we will only describe the final, correct model[7]. We will, however, reiterate that we did not make any changes to the model to 'ease' the proofs through: the levels of the model document design decisions only.

Issues regarding VDM's logic of partial functions and handling of 3-valued logic (undefined) values were handled with care, but informally. They should not be of concern for this problem, certainly not for our goals (of finding general proof strategies). They would be of concern for a general translation strategy from VDM to Isabelle.

[7] Chapter 4 discusses the evolution of the model.

Chapter 7

Conclusion

This report acts as a source document and summarises a case study in the use of verification tools to determine how realistic the ambition is of extracting the "why" from experts' use of such tools and provides a revision of an earlier description of an abstract model of an AI$_4$FM system that is linked to the case studies. We briefly discuss two important facets of our work below:

7.1 Patterns of proof

As noted above, there are many common patterns of proof in formal methods. Reification proofs, for example, always follow the same idea; furthermore, these proof patterns transfer across different formalisms, since they are often based around similar notions of refinement. Part of the goal in AI$_4$FM is to learn new proof patterns for domain problems (then reuse them in similar proofs). In the heap example, we developed strategies for proving lemmas about the separateness and disjointedness properties of the invariant. These strategies, though informal, were very useful and transferred , for example, from feasibility proofs of NEW to DISPOSE. Another important observation, though, is that some of the proof patterns (or strategies) used by an expert are more general and well-known (case analysis, for example), but the system needs to learn when and why to apply the pattern. As part of this project, we are attempting to catalogue some of these more general formal methods proof patterns, as we believe them to be of interest outside the project.

7.2 System to capture proof process

Capturing the interactive proof process and identifying the necessary abstractions –the expert's "whys"– is a cumbersome process if done without tool support. A large amount of proof process data presented in Chapter 3 can be captured and inferred automatically, by "listening" to the interactive proof and recording expert's insight. With this initial aim, the ProofProcess system has been developed alongside –and in support of– the interactive proof effort presented in this report. The system aims to facilitate proof analysis and strategy extraction.

The overall goal with the ProofProcess system is to develop a generic framework for capturing, analysing and inferring different proof processes. At the core, it provides a generic approach to represent proof processes.[1] The proposed "whys" and proof features provide high-level abstractions of the captured interactive proof. The system supports capturing the full history of

[1]The current model employed by the ProofProcess system corresponds to a subset of the abstract model presented in Chapter 3. The model focuses on representing proof process and currently lacks support for partitioning the data into bodies of knowledge or constructing the hierarchy of strategies.

formal proof development, including multiple (unfinished or alternative) proof attempts. Furthermore, the proofs can be recorded at variable granularity with different levels of abstraction. These features can be used in a generic manner and are applicable to proof processes from different theorem proving systems.

The generic core is extended with prover-specific integrations, creating the proof capture systems for particular theorem provers. These extensions provide prover-specific representations, such as the actual proof terms or proof commands used by the prover. Furthermore, they are responsible for integrating with the theorem prover –"wire-tapping" it– to record the low-level proof process details, such as expert interactions, provided proof commands, their results, associated proof context, used lemmas and other necessary information. A close integration records all activities in the prover, providing a full history of proof development. Currently, prototype ProofProcess extensions are available for two theorem proving systems: Isabelle [NPW02a] (via new Isabelle/Eclipse proof assistant[2]) and Z/EVES [Saa97] (via new Z/EVES integration with Community Z Tools[3]).

The ProofProcess tools have extensible and modular architecture. They are built on modern platforms of Eclipse[4] and EMF [SBPM08], using Java and Scala programming languages. The captured data is recorded as EMF objects and stored using the CDO framework.[5] This provides convenient data modelling functionality with a reliable embedded database solution. The storage implementation has been upgraded to cope with scaling issues arising when capturing industrial-type proofs, e.g. to achieve low memory usage and reasonable storage size. The implementation code for the ProofProcess framework and prover integrations is available as open-source.[6]

The captured proof process activities are subjected to analysis in order to try to infer certain information about the proof process automatically. Some of analysis techniques already available in the ProofProcess framework include recognition of proof re-runs, capturing backtracking and when a new proof attempt diverges, inferring basic proof structure (e.g. parallel case splits), finding certain kinds of important proof terms (e.g. identifying changed parts of the goals), etc. Furthermore, other high-level proof insight can be marked interactively by the expert, in order to indicate the important parts of the proof that "drive" the expert's decisions.

The main use of the captured proof process data is extracting reusable proof strategies. However, because the captured data presents a comprehensive account of the proof process with high-level proof descriptions, other uses and benefits are also expected. These include *proof maintenance*, *proof metrics*, *teaching and training* interactive theorem proving and others.

The architecture and implementation of the ProofProcess framework are part of the PhD research by the third author of this report [Vel14].

7.2.1 Future work

The research on "Rippling" [BBHI05b] will be incorporated into some future version of AI4FM. We believe that adding a *Why* of StuckInduction could trigger such proof failure analysis.

We are currently working on the tools for capturing and evaluating proof process (MWhy) (meta-)data (c.f. Chapter 3), and tools are available[7] as part of one of the authors' PhD [Vel14].

We have the database of MWhy data for both the Z/EVES and Isabelle proofs discussed here, but that is incomplete, and was used to drive tool development. Many (engineering) features were added and many more are still needed.

[2]Isabelle/Eclipse is available at `http://andriusvelykis.github.io/isabelle-eclipse`.

[3]Z/EVES integration via Community Z Tools (CZT) [MU05] and the new Eclipse-based IDE are part of the CZT 2.0 release. It is available at `http://czt.sourceforge.net`.

[4]Eclipse platform. `http://www.eclipse.org`

[5]Connected Data Objects (CDO) model repository. `http://www.eclipse.org/cdo`

[6]ProofProcess framework is available at `https://github.com/andriusvelykis/proofprocess`.

[7]`http://andrius.velykis.lt/` and `https://github.com/andriusvelykis/proofprocess`

We are also in the process of analysing the data with data mining algorithms, akin to the work done in [HK13].

Acknowledgements

It is a pleasure to acknowledge the fruitful collaboration with our Scottish colleagues in the AI$_4$FM project. The first author is grateful to Aaron Sloman for a useful discussion at the Birmingham BCTCS meeting. We also derived stimulus from discussions at the Schloß Dagstuhl event (12271) on "AI Meets Formal Software Development". Last, but by no means least, the authors are grateful for the EPSRC funding of the AI$_4$FM project.

Appendix A

General form of proof obligations (POs)

This summary of proof obligation templates comes from [Jon90, Appendix C].

A.1 Satisfiability

Each operation precondition needs to be strong enough to make the postcondition feasible. It is also known as feasibility proof.

$$\boxed{\text{satisfiable}}\ \frac{\overleftarrow{\sigma} \in \Sigma, i \in I \quad pre\text{-}OP(\overleftarrow{\sigma}, a)}{\exists \sigma \in \Sigma, o \in O \cdot post\text{-}OP(\overleftarrow{\sigma}, i, \sigma, o)}$$

A.2 Reification

When moving between data type representations (from level 0 sets to level 1 maps or set of pieces), we need to show that such a type jump keeps the properties of interest (i.e. types reify). To do that we need to define a retrieve relation (or function) mapping each representation, and then prove that their link is adequate. This is known as the adequacy proof.

$$\boxed{\text{adequacy}}\ \frac{\sigma_a \in \Sigma_a}{\exists \sigma_r \in \Sigma_r \cdot \sigma_a = retr(\sigma_r)}$$

We also need to show that, for every operation involved, the abstract (A) precondition needs to encompass the concrete (R) one. That is, the abstract precondition (or what you can assume) is wide enough to encompass all the cases discussed in the concrete precondition under the retrieve function mapping both data type domains. Adequacy proof is useful because it needs to be proved once and can be used on all operations involving the data types being refined.

In other words, the concrete operation preconditions can only assume as much as the abstract preconditions. This is known as the widening of the precondition and is defined next for every operation.

$$\boxed{\text{widen-pre}}\ \frac{\sigma_r \in \Sigma_r, i \in I \quad pre\text{-}OP\text{-}A(retr(\sigma_r), i)}{pre\text{-}OP\text{-}R(\sigma_r, i)}$$

Similarly, the concrete postcondition (or what is to be delivered) is within what was promised by the abstract postcondition. That is, using the assumption that the abstract precondition

holds, under the adequate retrieve function, the concrete postcondition is sufficient to establish the abstract postcondition contract.

$$\text{narrow-post}\ \frac{\begin{array}{l}\sigma_r, \overleftarrow{\sigma_r} \in \Sigma_r, i \in I, o \in O \\ \textit{pre-OP-A}(\textit{retr}(\overleftarrow{\sigma_r}), i) \\ \textit{post-OP-R}(\overleftarrow{\sigma_r}, i, \sigma_r, o)\end{array}}{\textit{post-OP-A}(\textit{retr}(\overleftarrow{\sigma_r}), i, \textit{retr}(\sigma_r), o)}$$

A.3 Sanity checks

Beyond satisfiability and reification of operations, it is also important to prove that our model actually reflect what we want/expect from a memory manager. These sanity checks can be state as conjectures to be proved at all levels in order to establish their usefulness in practice. Otherwise, we could have a feasible and (refinement) adequate model that does not do what the requirements/user wants.

For instance, it is desirable that *NEW* followed by *DISPOSE* on the same sizes is the identity memory. It is also desirable to enforce that *NEW*/*DISPOSE* shrink/grow memory accordingly with specific characteristics. The discussion in [JS90] does not include any sanity check proof obligations.

Appendix B

Isabelle formalisation nomenclature

B.1 The heap in Isabelle

B.1.1 Introduction

This section introduces the formal encoding of the heap storage case study in the Isabelle proof assistant. We do not introduce Isabelle in detail, but rather refer the reader to the Isabelle documentation [NPW02b, P$^+$94].

In the next section, we explain the naming conventions for our development and the overall architecture of the formalisation. Then, Section 5.3.1 describes the formalisation of level 0.

B.1.2 Background

We use locales to describe the VDM models of a Heap. This increases the modularity and clarity of the POs we are using Isabelle to prove, given of course the locale universally quantifying assumptions and preconditions.

For example, we can state:

lemma (**in** *LOCALE*) *Op1-FSB*:
$\exists \cdot$ *after-state result* . *invariant after-state* $\wedge$ *post after-state result*

We also use definition to capture VDM features. This is useful for the folding/unfolding of zooming pattern. For example, a property for an operation is stated as a definition:

definition
OP-N-X :: *STATE-N* $\Rightarrow$ *IN1* $\Rightarrow$ *INn* $\Rightarrow$ *bool*
where
OP-N-X S i-1 i-n $\equiv$ *pre-without-state-invariant-or-input-subtype S i-1 i-n*

and things like the invariant are also packaged up as definitions.

B.1.2.1 Naming conventions

We use the following conventions:

- *auxilliary functions are capitilised and have a "_" between each part of the name*
- *In the construction of the VDM operations macros, we introduce definitions of the following form, for each part of an operation and state. We use short names (pre, post, inv) for the various parts.*
  definition

STATE-N-inv :: *STATE-N* ⇒ *bool*
where
STATE-N-inv S-n ≡ *invariant S-n*

definition
OP-N-pre :: *STATE-N* ⇒ *IN1* ⇒ *INn* ⇒ *bool*
where
OP-N-pre S i-1 i-n ≡ *pre-without-state-invariant-or-input-subtype S i-1 i-n*

definition
OP-N-post :: *STATE-N* ⇒ *IN1* ⇒ *INn* ⇒ *STATE-N* ⇒ *Out* ⇒ *bool*
where
OP-N-post S i-1 i-n S′ out ≡ *post-without-state-invariant-or-IO-subtype S i-1 i-n S′ out*

- *We also introduce an Isabelle shortcut to unfold all the names that occur in a definition, as follows:*
  lemmas *OP-N-pre-defs* = *OP-N-pre-def OP-N-pre-OP1-def OP-N-pre-OP2-defetc.*
- *Finally, the specification of the VDM oerations themselves is given in a locale, where the inputs, invariant and preconditions are provided, and given long names. We use a locale* levelN_basic *to encode the common state and any common inputs and the invariant.*
  This is a useful construct as we also have common preconditions that arise in the translation of VDM types to Isabelle (and we need some predicates to enforce subtyping). This is discussed in more detail later.
  locale *level-N-basic* =
  fixes *f* :: *STATE-N* — common state
  and *s1* :: *IN1* — common inputs
  and *sn* :: *INn*
  assumes *l-N-input1-PROP*: *pred-input1-subtype*
  and *l-N-inputn-PROP*: *pred-inputn-subtype*
  and *l-N-invariant* : *STATE-inv f*

  locale *level-N-OP* = *level-N-basic* +
  fixes *i* :: *IN1* — specific inputs
  assumes *OP-precondition* : *OP-pre f s i* ∧ *STATE-inv fThe post-condition is then expressed as a definition within the locale:*
  definition (**in** *level-N-OP*)
  OP-N-postcondition :: *STATE-N* ⇒ *Out* ⇒ *bool*
  where
  OP-N-postcondition f′ r ≡ *OP-N-post f s1 sn f′ r* ∧ *STATE-N-inv f′*
- *Next, the proof obligations are specified using the following form and nomenclature:*
  definition (**in** *level-N-OP*)
  OP-N-feasibility :: *bool*
  where
  OP-N-feasibility ≡ (∃· *f′ r′* . *OP-N-postcondition f′ r′*)*which is then stated as a lemma:*
  lemma (**in** *level-N-OP*) *OP-N-Feasibility*: *OP-N-feasibility*

Appendix C

Proofs of Cliff's "that-lemma"

C.1 Procedural 'that lemma'

This is the Isabelle mechanisation of the proof in Section 3.2.3 on page 21.
theory *HEAP1CBJNoLemmas*
imports *HEAP1*
begin

lemma *l1*: *disjoint s1 s2* $\Longrightarrow$ *s3* $\subseteq$ *s2* $\Longrightarrow$ *disjoint s1 s3*
by (*metis Int-absorb2 Int-assoc Int-empty-right disjoint-def le-infI1 order-refl*)

lemma *l2v0*: *disjoint s1 s2* $\Longrightarrow$ *disjoint s2 s3* $\Longrightarrow$ *disjoint* (*s1* $\cup$ *s2*) *s3*
nitpick
oops
lemma *l2v1*: *s2* $\neq$ {} $\Longrightarrow$ *disjoint s1 s2* $\Longrightarrow$ *disjoint s2 s3* $\Longrightarrow$ *disjoint* (*s1* $\cup$ *s2*) *s3*
nitpick
oops

lemma *l2v2*: *s2* $\subseteq$ *s1* $\Longrightarrow$ *disjoint s1 s3* $\Longrightarrow$ *disjoint* (*s1* $\cup$ *s2*) *s3*
oops

lemma *l2*: *disjoint s1 s2* $\Longrightarrow$ *disjoint s1 s3* $\Longrightarrow$ *disjoint s2 s3* $\Longrightarrow$ *disjoint* (*s1* $\cup$ *s2*) *s3*
by (*metis Un-empty-left disjoint-def inf-sup-distrib2*)

lemma *l2o*: *disjoint s1 s3* $\Longrightarrow$ *disjoint s2 s3* $\Longrightarrow$ *disjoint* (*s1* $\cup$ *s2*) *s3*
apply (*metis Int-commute Un-empty-left disjoint-def inf-sup-distrib1*)
done

lemma *l3-1*: *nat1-map f* $\Longrightarrow$ *nat1-map*(*S* $-\!\lhd$ *f*)
by (*metis Diff-iff f-in-dom-ar-apply-subsume l-dom-dom-ar nat1-map-def*)

lemma *l3-2*: *l* $\in$ *dom* (*S* $-\!\lhd$ *f*) $\Longrightarrow$ *l* $\in$ *dom f*
unfolding *dom-antirestr-def*
by (*cases l*$\in$*S*, *auto*)

lemma *l3-3*: *l* $\in$ *dom* (*S* $-\!\lhd$ *f*) $\Longrightarrow$ *the* ((*S* $-\!\lhd$ *f*) *l*) = *the* (*f l*)
unfolding *dom-antirestr-def*
by (*cases l*$\in$*S*, *auto*)

lemma *l3*: *nat1-map f* $\Longrightarrow$ *locs*(*S* $-\!\lhd$ *f*) $\subseteq$ *locs f*
apply (*rule subsetI*)

unfolding *locs-def*
apply (*simp add*: *l3-1*)
apply (*erule bexE*)
apply (*frule l3-2*)
apply (*frule l3-3*,*simp*)
apply (*rule-tac x*=*s* **in** *bexI*)
by (*simp-all*)

lemma *l4*: *nat1 n* ⟹ *nat1 m* ⟹ *locs-of d* (*n*+*m*) = (*locs-of d n*) ∪ (*locs-of* (*d*+*n*) *m*)
unfolding *locs-of-def*
by *auto*

— New lemmas (relatively trivial)
lemma *l5*: *nat1-map f* ⟹ *x* ∈ *dom f* ⟹ *nat1* (*the*(*f x*))
by (*metis nat1-map-def*)

lemma *l6*: *nat1 y* ⟹ *y* < *s* ⟹ *locs-of* (*d*+*s*) *y* ⊆ *locs-of d s*
unfolding *locs-of-def*
apply *simp*
apply (*rule subsetI*)
find-theorems - ∈ {- . -}
apply (*elim conjE CollectE*)
apply (*intro conjI CollectI*)
apply (*simp*)
oops

lemma *l6-1*: *x* ∈ *dom f* ⟹ *nat1-map f* ⟹ *x* ∈ *locs-of x* (*the*(*f x*))
unfolding *locs-of-def*
apply (*frule l5*)
by *auto*

lemma *l6*: *x* ∈ *dom f* ⟹ *nat1-map f* ⟹ *x* ∈ *locs f*
unfolding *locs-def*
by (*metis UN-iff l6-1*)
— UNUSED, but discovered through the failure to prove l6 above, which led to change in l2v2

lemma *l7v0*: *d* ∈ *dom f* ⟹ *x* ∈ *locs-of d s* ⟹ *nat1-map f* ⟹ *x* ∈ *locs f*
unfolding *locs-def*
apply *simp*
oops

lemma *l7*: *d* ∈ *dom f* ⟹ *x* ∈ *locs-of d* (*the*(*f d*)) ⟹ *nat1-map f* ⟹ *x* ∈ *locs f*
unfolding *locs-def*
by (*simp*,*rule bexI*,*simp-all*)

— Going directly top bottom of proof - used wrong l2 lemma!
theorem *try1*: *F1-inv f* ⟹ *nat1 s* ⟹ *d*+*s* ∈ *dom f* ⟹ *disjoint* (*locs-of d s*) (*locs f*) ⟹
disjoint (*locs-of d* (*s*+ (*the*(*f*(*d*+*s*))))) (*locs* ({*d*+*s*} -⊲ *f*))
unfolding *F1-inv-def*
apply (*elim conjE*)
apply (*frule l3*[*of f* {*d*+*s*}]) — S4 : L3
apply (*frule l1*[*of locs-of d s locs f* (*locs* ({*d*+*s*} -⊲ *f*))],*simp*) — S5 : L1(S4,h)
— step 6 is strange: it is already what you want to conclude, yet it comes from h?
— here S6 comes from Disjoint f
oops

```
— Going in the order of steps
theorem try2:
      — h1 h2 h3 h4
      F1-inv f ⟹ nat1 s ⟹ d+s ∈ dom f ⟹ disjoint (locs-of d s) (locs f) ⟹
         disjoint (locs-of d (s+ (the(f(d+s))))) (locs ({d+s} -◁ f))
         unfolding F1-inv-def
         apply (elim conjE)
apply (frule l4[of s the(f(d+s)) d])  — S1 : L4(S2)
apply (rule l5,simp,simp)              — S2 : L5(h1[3],h3)
apply (erule ssubst)                   — infer : subs(S1) ; Nothing about S6
   thm  l2[of locs-of d s
               locs-of (d+s)
                 (s+(the(f(d+s))))
               locs ({d+s} -◁ f)]
apply (rule l2)                        — S3 : L2(S5, S6)
defer
   thm l1[of locs-of d s
             locs f
             locs ({d+s} -◁ f)]
      l3[of f {d+s}]
   pr
apply (frule l3[of f {d+s}])           — S4 : L3
apply (frule l1[of locs-of d s
                   locs f
               (locs ({d+s} -◁ f))],
               simp)                   — S5 : L1(S4,h)
— To me the backward steps towards the goal are harder to follow? How about S6? Will try backward

oops

— Just like try2 but going underneath disjoint definition
theorem try3:
      — h1 h2 h3 h4
      F1-inv f ⟹ nat1 s ⟹ d+s ∈ dom f ⟹ disjoint (locs-of d s) (locs f) ⟹
         disjoint (locs-of d (s+ (the(f(d+s))))) (locs ({d+s} -◁ f))
         unfolding F1-inv-def
         apply (elim conjE)
apply (frule l4[of s the(f(d+s)) d])  — S1 : L4(S2)
apply (rule l5,simp,simp)              — S2 : L5(h1[3],h3)
apply (erule ssubst)                   — infer : subs(S1) ; Nothing about S6
apply (rule l2)                        — S3 : L2(S5, S6)
defer
apply (rule l1[of - locs f -],simp)    — S5 : L1(S4,h4)
apply (rule l3,simp)                   — S4 : L3(h1[3])
defer
  — If I had a lemma (should create? no general enough?);
unfolding disjoint-def
  apply (simp add: disjoint-iff-not-equal)
  apply (intro ballI)
  apply (erule-tac x=x in ballE,simp-all)
  apply (erule-tac x=y in ballE,simp)
  apply (erule notE)
  apply (rule l7[of d+s f -],simp-all)

  apply (fold disjoint-def)
  apply (unfold disjoint-def)
```

```
  apply (simp add: disjoint-iff-not-equal)
  apply (frule l6,simp)
  apply (intro ballI)
  apply (frule l3-1[of f {d+s}])
  unfolding locs-def
  apply simp
  apply (elim bexE)
 thm l3 l3-1 l3-2 l5
  apply (frule l3-2)
  apply (simp add: l3-3)
  apply (frule l5[of - d+s],simp)
  apply (frule l5,simp) back
  apply (frule l5,simp) back back
  apply (erule ballE)+

  apply simp
  prefer 3
  apply (erule notE)
  unfolding locs-of-def
  apply simp
  nitpick
oops

— Version shown to Cliff - in step order and using l2 new
theorem try4:
      — h1 h2 h3 h4
     F1-inv f ⟹ nat1 s ⟹ d+s ∈ dom f ⟹ disjoint (locs-of d s) (locs f) ⟹
        disjoint (locs-of d (s+ (the(f(d+s))))) (locs ({d+s} -◁ f))
        unfolding F1-inv-def
        apply (elim conjE)
apply (frule l4[of s the(f(d+s)) d])  — S1 : L4(S2)
apply (rule l5,simp,simp)              — S2 : L5(h1[3],h3)
apply (erule ssubst)                   — infer : subs(S1) ; Nothing about S6
apply (rule l2)                        — S3 : L2(S5, S6)
defer
apply (rule l1[of - locs f -],simp)    — S5 : L1(S4,h4)
apply (rule l3,simp)                   — S4 : L3(h1[3])
defer
unfolding disjoint-def
  apply (simp add: disjoint-iff-not-equal)
  apply (intro ballI)
  apply (erule-tac x=x in ballE,simp-all)
  apply (erule-tac x=y in ballE,simp)
  apply (erule notE)
  apply (rule l7[of d+s f -],simp-all)
oops

lemma l3half-1: nat1-map f ⟹ (x ∈ locs f) = (∃·y ∈ dom f . x ∈ locs-of y (the(f y)))
unfolding locs-def
by (metis (mono-tags) UN-iff)

— Version shown to Cliff - in step order and using l2original + new lemma
lemma l3half:
— see lemma l_locs_dom_ar_iff:
  nat1-map f ⟹ Disjoint f ⟹ r ∈ dom f ⟹ locs({r} -◁ f) = locs f - locs-of r (the(f r))
apply (rule equalityI)
```

apply (*rule-tac* [*1*-] *subsetI*)
apply (*frule-tac* [*1*-] *l3-1*[*of* - {*r*}])
apply (*simp-all add*: *l3half-1*)
defer
apply (*elim conjE*)
defer
apply (*intro conjI*)
apply (*metis f-in-dom-ar-subsume f-in-dom-ar-the-subsume*)
apply (*erule-tac* [*1*-] *bexE*)
defer
apply (*rule-tac x*=*y* **in** *bexI*)
apply (*metis f-in-dom-ar-apply-not-elem singleton-iff*)
apply (*metis l-dom-dom-ar member-remove remove-def*)
apply (*frule f-in-dom-ar-subsume*)
apply (*frule f-in-dom-ar-the-subsume*)
unfolding *Disjoint-def disjoint-def Locs-of-def*
apply (*simp*)
by (*metis disjoint-iff-not-equal f-in-dom-ar-notelem*)

thm *f-in-dom-ar-subsume*
f-in-dom-ar-the-subsume
f-in-dom-ar-notelem
f-in-dom-ar-apply-not-elem
l-dom-dom-ar

lemma *l8*: *disjoint A* (*B* - *A*)
unfolding *disjoint-def*
by (*metis Diff-disjoint*)

— LATEST version from Cliff that avoids expanding locs def through lemmas (caveat: 3.5 is hard to prove
theorem *try7*:
— h1 h2 h3 h4
F1-inv f ⟹ *nat1 s* ⟹ *d*+*s* ∈ *dom f* ⟹ *disjoint* (*locs-of d s*) (*locs f*) ⟹
disjoint (*locs-of d* (*s*+ (*the*(*f*(*d*+*s*))))) (*locs* ({*d*+*s*} -◁ *f*))
unfolding *F1-inv-def*
apply (*elim conjE*)
apply (*frule l4*[*of s the*(*f*(*d*+*s*)) *d*]) — S1 : L4(S2)
apply (*rule l5*,*simp*,*simp*) — S2 : L5(h1[3],h3)
apply (*erule ssubst*) — infer : subs(S1) ; Nothing about S6
apply (*rule l2o*) — S3 : L2(S4, S6)
apply (*metis* (*full-types*) *l1 l3*)
by (*metis l3half l8*)

— trial lemma extracted from the last part of the next try proofs (try5/6 below)
lemma *trial*: *nat1-map f* ⟹ *Disjoint f* ⟹ *d*+*s* ∈ *dom f* ⟹ *disjoint* (*locs-of* (*d* + *s*) (*the* (*f* (*d* + *s*)))) (*locs* ({*d*+*s*} -◁ *f*))
unfolding *Disjoint-def Locs-of-def*
apply (*erule-tac x*=*d*+*s* **in** *ballE*) — S6 : S8
apply (*simp-all*)
unfolding *disjoint-def*
apply (*simp add*: *disjoint-iff-not-equal*)
apply (*intro ballI*)
unfolding *locs-def*
apply (*frule l3-1*[*of* - {*d*+*s*}])
apply *simp*

```
  apply (erule bexE)
  apply (frule l3-2)
  apply (frule f-in-dom-ar-notelem)
  apply (erule-tac x=sa in ballE,simp-all)
  apply (metis f-in-dom-ar-apply-subsume)
done

theorem try5:
      — h1 h2 h3 h4
      F1-inv f ⟹ nat1 s ⟹ d+s ∈ dom f ⟹ disjoint (locs-of d s) (locs f) ⟹
          disjoint (locs-of d (s+ (the(f(d+s))))) (locs ({d+s} -⊲ f))
          unfolding F1-inv-def
          apply (elim conjE)
apply (frule l4[of s the(f(d+s)) d])  — S1 : L4(S2)
apply (rule l5,simp,simp)              — S2 : L5(h1[3],h3)
apply (erule ssubst)                   — infer : subs(S1) ; Nothing about S6
apply (rule l2o)                       — S3 : L2(S4, S6)
apply (rule l1[of - locs f -],simp)    — S4 : L1(S5,h4)
apply (rule l3,simp)                   — S5 : L3(h1[3])

apply (frule l3half,simp,simp,simp)      — S8 : L3.5(h1[1])

oops

theorem try6:
      — h1 h2 h3 h4
      F1-inv f ⟹ nat1 s ⟹ d+s ∈ dom f ⟹ disjoint (locs-of d s) (locs f) ⟹
          disjoint (locs-of d (s+ (the(f(d+s))))) (locs ({d+s} -⊲ f))
          unfolding F1-inv-def
          apply (elim conjE)
apply (frule l4[of s the(f(d+s)) d])  — S1 : L4(S2)
apply (rule l5,simp,simp)              — S2 : L5(h1[3],h3)
apply (erule ssubst)                   — infer : subs(S1) ; Nothing about S6
apply (rule l2o)                       — S3 : L2(S4, S6)
apply (rule l1[of - locs f -],simp)    — S4 : L1(S5,h4)
apply (rule l3,simp)                   — S5 : L3(h1[3])
apply (rule trial,simp,simp,simp)
done

end
```

C.2 Isar 'that lemma'

This is an Isar-style mechanisation of the proof in Section 3.2.3 on page 21.

```
lemma L1:
  assumes disjoint s1 s2
  and s3 ⊆ s2
  shows disjoint s1 s3
  using assms unfolding disjoint-def
  by blast

lemma L1pt5:
  shows disjoint s2 (s1 - s2)
    unfolding disjoint-def by simp
```

lemma *L2*:
  assumes *disjoint s1 s3*
  and *disjoint s2 s3*
  shows *disjoint* (*s1* ∪ *s2*) *s3*
  using *assms* **unfolding** *disjoint-def*
  by *blast*

lemma *L3*:
 assumes ∗: *nat1-map f*
 shows *locs* (*S* -◁ *f*) ⊆ *locs f*
proof
 fix *x* **assume** *xin-domar*: *x* ∈ *locs* (*S* -◁ *f*)
 then have *x* ∈ (⋃ *s*∈*dom* (*S* -◁ *f*). *locs-of s* (*the* ((*S* -◁ *f*) *s*)))
  by (*simp add*: *locs-def* ∗ *dom-ar-nat1-map*)
 then have *x* ∈ (⋃ *s*∈*dom f*. *locs-of s* (*the* (*f s*)))
  by (*smt UN-iff f-in-dom-ar-apply-not-elem l-dom-ar-notin-dom-or*)
 thus *x* ∈ *locs f* **by** (*simp add*: *locs-def* ∗)
qed

lemma *L3pt5*:
assumes *s*∈ *dom f*
and *Disjoint f*
and *nat1-map f*
shows *locs* ({*s*}-◁ *f*) = *locs f* - *locs-of s* (*the* (*f s*))
using *assms* **by** (*simp add*: *l-locs-of-dom-ar*)

lemma *L4*: *nat1 n* ⟹ *nat1 m* ⟹ *locs-of d* (*n*+*m*) = (*locs-of d n*) ∪ (*locs-of* (*d*+*n*) *m*)
unfolding *locs-of-def*
by *auto*

lemma *that-lemma*:
 assumes *a1*: *F1-inv f*
 and *a2*: *disjoint* (*locs-of d s*) (*locs f*)
 and *a3*: *d*+*s* ∈ *dom f*
 and *a4*: *nat1 s*
 shows *disjoint* (*locs-of d* (*s* + (*the* (*f* (*d*+*s*)))))
              (*locs* ({*d*+*s*} -◁ *f*))
proof -
 from *a1* **show** *?thesis*
 proof
 assume *Disj*: *Disjoint f*
 and *n1map*: *nat1-map f*
 show *?thesis* — Standard set-up of the problem complete
 proof(*subst L4*) — Step 1: backwards application of L4
  show *nat1 s* **by** (*rule a4*) — Direct from assm
 next
  show *nat1* (*the* (*f* (*d* + *s*))) — Step 2: solved (almost) directly from our hyp
   using *n1map nat1-map-def a3* **by** *simp*
 next — Resulting goal
  show *disjoint* (*locs-of d s* ∪ *locs-of* (*d* + *s*) (*the* (*f* (*d* + *s*))))
              (*locs* ({*d* + *s*} -◁ *f*))
  proof(*rule L2*) — Step 3: backward application of L2
   from *a2* **show** *disjoint* (*locs-of d s*) (*locs* ({*d*+*s*} -◁*f*)) — Step 4
   proof (*rule L1*)
    from *n1map* **show** *locs* ({*d* + *s*} -◁ *f*) ⊆ *locs f* — Step 5
     by(*rule L3*)

```
      qed
     next
      show (disjoint (locs-of (d + s) (the (f (d + s))))
                     (locs ({d + s} -⊲ f))
      proof (subst L3pt5) — Step 6: Substition of L3.5
        from a3 Disj n1map show d + s ∈ dom f Disjoint f nat1-map f
         by simp-all
       next
        show (disjoint (locs-of (d + s) (the (f (d + s))))
            (locs f - locs-of (d + s) (the (f (d + s)))))
         by (rule L1pt5) — Step 7
      qed
    qed
  qed
qed
qed
```

Appendix D

VDM Maps auxiliary library

theory *VDMMaps*
imports *Main*
begin

ML ⟨⟨ *quick-and-dirty* := *true* ⟩⟩

D.1 Extra map operators

definition
  dom-restr :: *'a set* ⇒ (*'a* ⇀ *'b*) ⇒ (*'a* ⇀ *'b*) (**infixr** ◁ *110*)
where
  [*intro*!]: *s* ◁ *m* ≡ *m* |` *s*

definition
  ran-restr :: (*'a* ⇀ *'b*) ⇒ *'b set* ⇒ (*'a* ⇀ *'b*) (**infixl** ▷ *105*)
where
  m ▷ *s* ≡ (λ*x* . *if* (∃· *y*. *m x* = *Some y* ∧ *y* ∈ *s*) *then m x else None*)

definition
  dom-antirestr :: *'a set* ⇒ (*'a* ⇀ *'b*) ⇒ (*'a* ⇀ *'b*) (**infixr** -◁ *110*)
where
  s -◁ *m* ≡ (λ*x*. *if x* : *s then None else m x*)

definition
  ran-antirestr :: (*'a* ⇀ *'b*) ⇒ *'b set* ⇒ (*'a* ⇀ *'b*) (**infixl** ▷- *105*)
where
  m ▷- *s* ≡ (λ*x* . *if* (∃· *y*. *m x* = *Some y* ∧ *y* ∈ *s*) *then None else m x*)

definition
  dagger :: (*'a* ⇀ *'b*) ⇒ (*'a* ⇀ *'b*) ⇒ (*'a* ⇀ *'b*) (**infixl** † *100*)
where
  [*intro*!]: *f* † *g* ≡ *f* ++ *g*

definition
  munion :: (*'a* ⇀ *'b*) ⇒ (*'a* ⇀ *'b*) ⇒ (*'a* ⇀ *'b*) (**infixl** ∪*m* *90*)
where
  [*intro*!]: *f* ∪*m* *g* ≡ (*if dom f* ∩ *dom g* = {} *then f* † *g else undefined*)

And by the way, this use of Isabelle's undefined value is a bit of a cheeky cheat. It basically

means we shouldn't get to undefined, rather than we are handling undefinedness. That's because the value is comparable (see next lemma). In effect, if we ever reach undefined it means we have some partial function application outside its domain somewhere within any rewriting chain. As one cannot reason about this value, it can be seen as a flag for an error to be avoided.

D.2 Set operators lemmas

lemma *l-psubset-insert*: $x \notin S \Longrightarrow S \subset \mathit{insert}\ x\ S$
by *blast*

lemma *l-right-diff-left-dist*: $S - (T - U) = (S - T) \cup (S \cap U)$
by (*metis Diff-Compl Diff-Int diff-eq*)
thm *Diff-Compl*
Diff-Int
diff-eq

lemma *l-diff-un-not-equal*: $R \subset T \Longrightarrow T \subseteq S \Longrightarrow S - T \cup R \neq S$
by *auto*

D.3 Map operators lemmas

lemma *l-map-non-empty-has-elem-conv*:
$g \neq \mathit{empty} \longleftrightarrow (\exists \cdot x\ .\ x \in \mathit{dom}\ g)$
by (*metis domIff*)

lemma *l-map-non-empty-dom-conv*:
$g \neq \mathit{empty} \longleftrightarrow \mathit{dom}\ g \neq \{\}$
by (*metis dom-eq-empty-conv*)

lemma *l-map-non-empty-ran-conv*:
$g \neq \mathit{empty} \longleftrightarrow \mathit{ran}\ g \neq \{\}$
by (*metis empty-iff equals0I*
fun-upd-triv option.exhaust
ranI ran-restrictD restrict-complement-singleton-eq)

D.3.0.2 Domain restriction weakening lemmas [EXPERT]

lemma *l-dom-r-iff*: $\mathit{dom}(S \lhd g) = S \cap \mathit{dom}\ g$
by (*metis Int-commute dom-restr-def dom-restrict*)

lemma *l-dom-r-subset*: $(S \lhd g) \subseteq_m g$
by (*metis Int-iff dom-restr-def l-dom-r-iff map-le-def restrict-in*)

lemma *l-dom-r-accum*: $S \lhd (T \lhd g) = (S \cap T) \lhd g$
by (*metis Int-commute dom-restr-def restrict-restrict*)

lemma *l-dom-r-nothing*: $\{\} \lhd f = \mathit{empty}$
by (*metis dom-restr-def restrict-map-to-empty*)

lemma *l-dom-r-empty*: $S \lhd \mathit{empty} = \mathit{empty}$
by (*metis dom-restr-def restrict-map-empty*)

lemma *l-dom-r-nothing-empty*: $S = \{\} \Longrightarrow S \lhd f = Map.empty$
by (*metis l-dom-r-nothing*)

lemma *f-in-dom-r-apply-elem*: $x \in S \Longrightarrow ((S \lhd f)\ x) = (f\ x)$
by (*metis dom-restr-def restrict-in*)

lemma *f-in-dom-r-apply-the-elem*: $x \in dom\ f \Longrightarrow x \in S \Longrightarrow ((S \lhd f)\ x) = Some(the(f\ x))$
by (*metis domD f-in-dom-r-apply-elem the.simps*)

lemma *l-dom-r-disjoint-weakening*: $A \cap B = \{\} \Longrightarrow dom(A \lhd f) \cap dom(B \lhd f) = \{\}$
by (*metis dom-restr-def dom-restrict inf-bot-right inf-left-commute restrict-restrict*)

lemma *l-dom-r-subseteq*: $S \subseteq dom\ f \Longrightarrow dom\ (S \lhd f) = S$ **unfolding** *dom-restr-def*
by (*metis Int-absorb1 dom-restrict*)

lemma *l-dom-r-dom-subseteq*: $(dom\ (\ S \lhd f)) \subseteq dom\ f$
unfolding *dom-restr-def* **by** *auto*

lemma *l-the-dom-r*: $x \in dom\ f \Longrightarrow x \in S \Longrightarrow the\ ((\ S \lhd f)\ x) = the\ (f\ x)$
by (*metis f-in-dom-r-apply-elem*)

lemma *l-in-dom-dom-r*: $x \in dom\ (S \lhd f) \Longrightarrow x \in S$
by (*metis Int-iff l-dom-r-iff*)

lemma *l-dom-r-singleton*: $x \in dom\ f \Longrightarrow (\{x\} \lhd f) = [x \mapsto the\ (f\ x)]$
unfolding *dom-restr-def*
by *auto*

lemma *singleton-map-dom*:
assumes $dom\ f = \{x\}$ **shows** $f = [x \mapsto the\ (f\ x)]$
proof -
from *assms* **obtain** y **where** $f = [x \mapsto y]$
by (*metis dom-eq-singleton-conv*)
then have $y = the\ (f\ x)$ **by** (*metis fun-upd-same the.simps*)
thus *?thesis* **by** (*metis* ‹$f = [x \mapsto y]$›)
qed

lemmas *restr-simps* = *l-dom-r-iff l-dom-r-accum l-dom-r-nothing l-dom-r-empty*
f-in-dom-r-apply-elem l-dom-r-disjoint-weakening l-dom-r-subseteq
l-dom-r-dom-subseteq

D.3.0.3 Domain anti restriction weakening lemmas [EXPERT]

lemma *f-in-dom-ar-subsume*: $l \in dom\ (S \mathbin{-\!\lhd} f) \Longrightarrow\ l \in dom\ f$
unfolding *dom-antirestr-def*
by (*cases* $l \in S$, *auto*)

lemma *f-in-dom-ar-notelem*: $l \in dom\ (\{r\} \mathbin{-\!\lhd} f) \Longrightarrow l \neq r$
unfolding *dom-antirestr-def*

by *auto*

lemma *f-in-dom-ar-the-subsume*:
$l \in dom\ (S\ \text{-}\!\lhd\ f) \Longrightarrow the\ ((S\ \text{-}\!\lhd\ f)\ l) = the\ (f\ l)$
unfolding *dom-antirestr-def*
by (*cases* $l \in S$, *auto*)

lemma *f-in-dom-ar-apply-subsume*:
$l \in dom\ (S\ \text{-}\!\lhd\ f) \Longrightarrow ((S\ \text{-}\!\lhd\ f)\ l) = (f\ l)$
unfolding *dom-antirestr-def*
by (*cases* $l \in S$, *auto*)

lemma *f-in-dom-ar-apply-not-elem*: $l \notin S \Longrightarrow (S\ \text{-}\!\lhd\ f)\ l = f\ l$
by (*metis dom-antirestr-def*)

lemma *f-dom-ar-subset-dom*:
$dom(S\ \text{-}\!\lhd\ f) \subseteq dom\ f$
unfolding *dom-antirestr-def dom-def*
by *auto*

lemma *l-dom-dom-ar*:
$dom(S\ \text{-}\!\lhd\ f) = dom\ f\ -\ S$
unfolding *dom-antirestr-def*
by (*smt Collect-cong domIff dom-def set-diff-eq*)

lemma *l-dom-ar-accum*:
$S\ \text{-}\!\lhd\ (T\ \text{-}\!\lhd\ f) = (S \cup T)\ \text{-}\!\lhd\ f$
unfolding *dom-antirestr-def*
by *auto*

lemma *l-dom-ar-nothing*:
$S \cap dom\ f = \{\} \Longrightarrow S\ \text{-}\!\lhd\ f = f$
unfolding *dom-antirestr-def*
apply (*simp add*: *fun-eq-iff*)
by (*metis disjoint-iff-not-equal domIff*)

lemma *l-dom-ar-empty-lhs*:
$\{\}\ \text{-}\!\lhd\ f = f$
by (*metis Int-empty-left l-dom-ar-nothing*)

lemma *l-dom-ar-empty-rhs*:
$S\ \text{-}\!\lhd\ empty = empty$
by (*metis Int-empty-right dom-empty l-dom-ar-nothing*)

lemma *l-dom-ar-everything*:
  $dom\ f \subseteq S \Longrightarrow S \lhd\!\!- f = empty$
by (*metis domIff dom-antirestr-def in-mono*)

lemma *l-map-dom-ar-subset*: $S \lhd\!\!- f \subseteq_m f$
by (*metis domIff dom-antirestr-def map-le-def*)

lemma *l-dom-ar-none*: $\{\} \lhd\!\!- f = f$
unfolding *dom-antirestr-def*
by (*simp add*: *fun-eq-iff*)

lemma *l-map-dom-ar-neq*: $S \subseteq dom\ f \Longrightarrow S \neq \{\} \Longrightarrow S \lhd\!\!- f \neq f$
apply (*subst fun-eq-iff*)
apply (*insert ex-in-conv*[*of S*])
apply *simp*
apply (*erule exE*)
unfolding *dom-antirestr-def*
apply (*rule exI*)
apply *simp*
apply (*intro impI conjI*)
apply *simp-all*
by (*metis domIff set-mp*)

lemma *l-dom-ar-not-in-dom*:
  assumes $*$: $x \notin dom\ f$
  shows $x \notin dom\ (s \lhd\!\!- f)$
by (*metis* $*$ *domIff dom-antirestr-def*)

lemma *l-dom-ar-not-in-dom2*: $x \in F \Longrightarrow x \notin dom\ (F \lhd\!\!- f)$
by (*metis domIff dom-antirestr-def*)

lemma *l-dom-ar-notin-dom-or*: $x \notin dom\ f \vee x \in S \Longrightarrow x \notin dom\ (S \lhd\!\!- f)$
by (*metis Diff-iff l-dom-dom-ar*)

lemma *l-in-dom-ar*: $x \notin F \Longrightarrow x \in dom\ f \Longrightarrow x \in dom\ \ (F \lhd\!\!- f)$
by (*metis f-in-dom-ar-apply-not-elem domIff*)

lemma *l-dom-ar-insert*: $((insert\ x\ F) \lhd\!\!- f) = \{x\} \lhd\!\!- (F \lhd\!\!- f)$
proof
  fix *xa*
  show $(insert\ x\ F \lhd\!\!- f)\ xa = (\{x\} \lhd\!\!- F \lhd\!\!- f)\ xa$
  apply (*cases x*= *xa*)
  apply (*simp add*: *dom-antirestr-def*)
  apply (*cases xa*$\in$*F*)
  apply (*simp add*: *dom-antirestr-def*)

apply (*subst f-in-dom-ar-apply-not-elem*)
apply *simp*
apply (*subst f-in-dom-ar-apply-not-elem*)
apply *simp*
apply (*subst f-in-dom-ar-apply-not-elem*)
apply *simp*
apply *simp*
done
qed

lemma *l-dom-ar-absorb-singleton*: $x \in F \Longrightarrow (\{x\} \text{ -}\lhd F \text{ -}\lhd f) = (F \text{ -}\lhd f)$
by (*metis l-dom-ar-insert insert-absorb*)

lemma *l-dom-ar-disjoint-weakening*:
$dom\ f \cap Y = \{\} \Longrightarrow dom\ (X \text{ -}\lhd f) \cap Y = \{\}$
by (*metis Diff-Int-distrib2 empty-Diff l-dom-dom-ar*)

lemma *l-dom-ar-singletons-comm*: $\{x\}\text{-}\lhd \{y\} \text{ -}\lhd f = \{y\}\text{-}\lhd \{x\} \text{ -}\lhd f$
by (*metis l-dom-ar-insert insert-commute*)

lemmas *antirestr-simps = f-in-dom-ar-subsume f-in-dom-ar-notelem f-in-dom-ar-the-subsume f-in-dom-ar-apply-subsume f-in-dom-ar-apply-not-elem f-dom-ar-subset-dom l-dom-dom-ar l-dom-ar-accum l-dom-ar-nothing l-dom-ar-empty-lhs l-dom-ar-empty-rhs l-dom-ar-everything l-dom-ar-none l-dom-ar-not-in-dom l-dom-ar-not-in-dom2 l-dom-ar-notin-dom-or l-in-dom-ar l-dom-ar-disjoint-weakening*

D.3.0.4 Map override weakening lemmas [EXPERT]

lemma *l-dagger-assoc*:
$f \dagger (g \dagger h) = (f \dagger g) \dagger h$
by (*metis dagger-def map-add-assoc*)
thm *ext option.split fun-eq-iff*

lemma *l-dagger-apply*:
$(f \dagger g)\ x = (if\ x \in dom\ g\ then\ (g\ x)\ else\ (f\ x))$
unfolding *dagger-def*
by (*metis* (*full-types*) *map-add-dom-app-simps*(*1*) *map-add-dom-app-simps*(*3*))

lemma *l-dagger-dom*:
$dom(f \dagger g) = dom\ f \cup dom\ g$
unfolding *dagger-def*
by (*metis dom-map-add sup-commute*)

lemma *l-dagger-lhs-absorb*:
$dom\ f \subseteq dom\ g \Longrightarrow f \dagger g = g$
apply (*rule ext*)
by(*metis dagger-def l-dagger-apply map-add-dom-app-simps*(*2*) *set-rev-mp*)

lemma *l-dagger-lhs-absorb-ALT-PROOF*:

dom f ⊆ dom g ⟹ f † g = g
apply (*rule ext*)
apply (*simp add*: *l-dagger-apply*)
apply (*rule impI*)
find-theorems - *∉* - *⟹* - *name*:*Set*
apply (*drule contra-subsetD*)
unfolding *dom-def*
by (*simp-all*)

lemma *l-dagger-empty-lhs*:
empty † f = f
by (*metis dagger-def empty-map-add*)

lemma *l-dagger-empty-rhs*:
f † empty = f
by (*metis dagger-def map-add-empty*)

lemma *dagger-notemptyL*: *f ≠ empty ⟹ f † g ≠ empty* **by** (*metis dagger-def map-add-None*)

lemma *dagger-notemptyR*: *g ≠ empty ⟹ f † g ≠ empty* **by** (*metis dagger-def map-add-None*)

lemma *l-dagger-dom-ar-assoc*:
S ∩ dom g = {} ⟹ (S -⊲ f) † g = S -⊲ (f † g)
apply (*simp add*: *fun-eq-iff*)
apply (*simp add*: *l-dagger-apply*)
apply (*intro allI impI conjI*)
unfolding *dom-antirestr-def*
apply (*simp-all add*: *l-dagger-apply*)
by (*metis dom-antirestr-def l-dom-ar-nothing*)
thm *map-add-comm*

lemma *l-dagger-not-empty*:
g ≠ empty ⟹ f † g ≠ empty
by (*metis dagger-def map-add-None*)

lemma *in-dagger-domL*:
x ∈ dom f ⟹ x ∈ dom(f † g)
by (*metis dagger-def domIff map-add-None*)

lemma *in-dagger-domR*:
x ∈ dom g ⟹ x ∈ dom(f † g)
by (*metis dagger-def domIff map-add-None*)

lemma *the-dagger-dom-right*:
assumes *x ∈ dom g*
shows *the ((f † g) x) = the (g x)*
by (*metis assms dagger-def map-add-dom-app-simps*(*1*))

lemma *the-dagger-dom-left*:
assumes $x \notin dom\ g$
shows $the\ ((f \dagger g)\ x) = the\ (f\ x)$
by (*metis assms dagger-def map-add-dom-app-simps*(*3*))

lemma *the-dagger-mapupd-dom*: $x \neq y \Longrightarrow (f \dagger [y \mapsto z])\ x = f\ x$
by (*metis dagger-def fun-upd-other map-add-empty map-add-upd*)

lemma *dagger-upd-dist*: $f \dagger fa(e \mapsto r) = (f \dagger fa)(e \mapsto r)$ **by** (*metis dagger-def map-add-upd*)

lemma *antirestr-then-dagger-notin*: $x \notin dom\ f \Longrightarrow \{x\} \lhd (f \dagger [x \mapsto y]) = f$
proof
fix z
assume $x \notin dom\ f$
show $(\{x\} \lhd (f \dagger [x \mapsto y]))\ z = f\ z$
by (*metis* ‹$x \notin dom\ f$› *domIff dom-antirestr-def fun-upd-other insertI1 l-dagger-apply singleton-iff*)

qed
lemma *antirestr-then-dagger*: $r \in dom\ f \Longrightarrow \{r\} \lhd f \dagger [r \mapsto the\ (f\ r)] = f$
proof
fix x
assume $*$: $r \in dom\ f$
show $(\{r\} \lhd f \dagger [r \mapsto the\ (f\ r)])\ x = f\ x$
proof (*subst l-dagger-apply*,*simp*,*intro conjI impI*)
assume $x = r$ **then show** $Some\ (the\ (f\ r)) = f\ r$ **using** $*$ **by** *auto*
next
assume $x \neq r$ **then show** $(\{r\} \lhd f)\ x = f\ x$ **by** (*metis f-in-dom-ar-apply-not-elem singleton-iff*)
qed
qed

lemma *dagger-notin-right*: $x \notin dom\ g \Longrightarrow (f \dagger g)\ x = f\ x$
by (*metis l-dagger-apply*)

lemma *dagger-notin-left*: $x \notin dom\ f \Longrightarrow (f \dagger g)\ x = g\ x$
by (*metis dagger-def map-add-dom-app-simps*(*2*))

lemma *l-dagger-commute*: $dom\ f \cap dom\ g = \{\} \Longrightarrow f \dagger g = g \dagger f$
unfolding *dagger-def*
apply (*rule map-add-comm*)
by *simp*

lemmas *dagger-simps* = *l-dagger-assoc l-dagger-apply l-dagger-dom l-dagger-lhs-absorb l-dagger-empty-lhs l-dagger-empty-rhs dagger-notemptyL dagger-notemptyR l-dagger-not-empty in-dagger-domL in-dagger-domR the-dagger-dom-right the-dagger-dom-left the-dagger-mapupd-dom dagger-upd-dist antirestr-then-dagger-notin antirestr-then-dagger dagger-notin-right dagger-notin-left*

D.3.0.5 Map update weakening lemmas [EXPERT]

without the condition nitpick finds counter example

lemma *l-inmapupd-dom-iff*:
$l \neq x \Longrightarrow (l \in dom\ (f(x \mapsto y))) = (l \in dom\ f)$
by (*metis* (*full-types*) *domIff fun-upd-apply*)

lemma *l-inmapupd-dom*:
$l \in dom\ f \Longrightarrow l \in dom\ (f(x \mapsto y))$
by (*metis dom-fun-upd insert-iff option.distinct*(*1*))

lemma *l-dom-extend*:
$x \notin dom\ f \Longrightarrow\ dom\ (f1(x \mapsto y)) = dom\ f1 \cup \{x\}$
by *simp*

lemma *l-updatedom-eq*:
$x{=}l \Longrightarrow the\ ((f(x \mapsto the\ (f\ x)\ \text{-}\ s))\ l) = the\ (f\ l)\ \text{-}\ s$
by *auto*

lemma *l-updatedom-neq*:
$x{\neq}l \Longrightarrow the\ ((f(x \mapsto the\ (f\ x)\ \text{-}\ s))\ l) = the\ (f\ l)$
by *auto*

— A helper lemma to have map update when domain is updated
lemma *l-insertUpdSpec-aux*: $dom\ f = insert\ x\ F \Longrightarrow (f0 = (f\ |`\ F)) \Longrightarrow f = f0\ (x \mapsto the\ (f\ x))$
proof *auto*
assume *insert*: $dom\ f = insert\ x\ F$
then have $x \in dom\ f$ **by** *simp*
then show $f = (f\ |`\ F)(x \mapsto the\ (f\ x))$ **using** *insert*
unfolding *dom-def*
apply *simp*
apply (*rule ext*)
apply *auto*
done
qed

lemma *l-the-map-union-right*: $x \in dom\ g \Longrightarrow dom\ f \cap dom\ g = \{\} \Longrightarrow the\ ((f \cup m\ g)\ x) = the\ (g\ x)$
by (*metis l-dagger-apply munion-def*)

lemma *l-the-map-union-left*: $x \in dom\ f \Longrightarrow dom\ f \cap dom\ g = \{\} \Longrightarrow the\ ((f \cup m\ g)\ x) = the\ (f\ x)$
by (*metis l-dagger-apply l-dagger-commute munion-def*)

lemmas *upd-simps* = *l-inmapupd-dom-iff l-inmapupd-dom l-dom-extend*
l-updatedom-eq l-updatedom-neq

D.3.0.6 Map union (VDM-specific) weakening lemmas [EXPERT]

lemma *k-munion-map-upd-wd*:
$x \notin dom\ f \Longrightarrow dom\ f \cap dom\ [x \mapsto y] = \{\}$
by (*metis Int-empty-left Int-insert-left dom-eq-singleton-conv inf-commute*)

lemma *l-munion-apply*:
$dom\ f \cap dom\ g = \{\} \Longrightarrow (f \cup m\ g)\ x = (if\ x \in dom\ g\ then\ (g\ x)\ else\ (f\ x))$
unfolding *munion-def*

by (*simp add*: *l-dagger-apply*)

lemma *l-munion-dom*:
dom f ∩ *dom g* = {} ⟹ *dom*(*f* ∪*m g*) = *dom f* ∪ *dom g*
unfolding *munion-def*
by (*simp add*: *l-dagger-dom*)

lemma *b-dagger-munion-aux*:
dom(*dom g* -◁ *f*) ∩ *dom g* = {}
apply (*simp add*: *l-dom-dom-ar*)
by (*metis Diff-disjoint inf-commute*)

lemma *b-dagger-munion*:
(*f* † *g*) = (*dom g* -◁ *f*) ∪*m g*
find-theorems (*300*) - = (-::(- ⇒ -)) -*name*:*Predicate* -*name*:*Product* -*name*:*Quick* -*name*:*New* -*name*:*Record* -*name*:*Quotient*
-*name*:*Hilbert* -*name*:*Nitpick* -*name*:*Random* -*name*:*Transitive* -*name*:*Sum-Type* -*name*:*DSeq* -*name*:*Datatype* -*name*:*Enum*
-*name*:*Big* -*name*:*Code* -*name*:*Divides*
thm *fun-eq-iff*[*of f* † *g* (*dom g* -◁ *f*) ∪*m g*]
apply (*simp add*: *fun-eq-iff*)
apply (*simp add*: *l-dagger-apply*)
apply (*cut-tac b-dagger-munion-aux*[*of g f*])
apply (*intro allI impI conjI*)
apply (*simp-all add*: *l-munion-apply*)
unfolding *dom-antirestr-def*
by *simp*

lemma *l-munion-assoc*:
dom f ∩ *dom g* = {} ⟹ *dom g* ∩ *dom h* = {} ⟹ (*f* ∪*m g*) ∪*m h* = *f* ∪*m* (*g* ∪*m h*)
unfolding *munion-def*
apply (*simp add*: *l-dagger-dom*)
apply (*intro conjI impI*)
apply (*metis l-dagger-assoc*)
apply (*simp-all add*: *disjoint-iff-not-equal*)
apply (*erule-tac* [*1*-] *bexE*)
apply *blast*
apply *blast*
done

lemma *l-munion-commute*:
dom f ∩ *dom g* = {} ⟹ *f* ∪*m g* = *g* ∪*m f*
by (*metis b-dagger-munion l-dagger-commute l-dom-ar-nothing munion-def*)

lemma *l-munion-subsume*:
x ∈ *dom f* ⟹ *the*(*f x*) = *y* ⟹ *f* = ({*x*} -◁ *f*) ∪*m* [*x* ↦ *y*]
apply (*subst fun-eq-iff*)
apply (*intro allI*)
apply (*subgoal-tac dom*({*x*} -◁ *f*) ∩ *dom* [*x* ↦ *y*] = {})
apply (*simp add*: *l-munion-apply*)
apply (*metis domD dom-antirestr-def singletonE the.simps*)
by (*metis Diff-disjoint Int-commute dom-eq-singleton-conv l-dom-dom-ar*)*Perhaps add g* ⊆*m f instead?*
lemma *l-munion-subsumeG*:

$dom\ g \subseteq dom\ f \Longrightarrow \forall \cdot x \in dom\ g\ .\ f\ x = g\ x \Longrightarrow f = (dom\ g \mathbin{-\!\!\lhd} f) \cup m\ g$

unfolding *munion-def*
apply (*subgoal-tac dom* $(dom\ g \mathbin{-\!\!\lhd} f) \cap dom\ g = \{\}$)
apply *simp*
apply (*subst fun-eq-iff*)
apply (*rule allI*)
apply (*simp add*: *l-dagger-apply*)
apply (*intro conjI impI*)+
unfolding *dom-antirestr-def*
apply (*simp*)
apply (*fold dom-antirestr-def*)
by (*metis Diff-disjoint inf-commute l-dom-dom-ar*)

lemma *l-munion-dom-ar-assoc*:
$S \subseteq dom\ f \Longrightarrow dom\ f \cap dom\ g = \{\} \Longrightarrow (S \mathbin{-\!\!\lhd} f) \cup m\ g = S \mathbin{-\!\!\lhd} (f \cup m\ g)$
unfolding *munion-def*
apply (*subgoal-tac dom* $(S \mathbin{-\!\!\lhd} f) \cap dom\ g = \{\}$)
defer *1*
apply (*metis Diff-Int-distrib2 empty-Diff l-dom-dom-ar*)
apply *simp*
apply (*rule l-dagger-dom-ar-assoc*)
by (*metis equalityE inf-mono subset-empty*)

lemma *l-munion-empty-rhs*:
$(f \cup m\ empty) = f$
unfolding *munion-def*
by (*metis dom-empty inf-bot-right l-dagger-empty-rhs*)

lemma *l-munion-empty-lhs*:
$(empty \cup m\ f) = f$
unfolding *munion-def*
by (*metis dom-empty inf-bot-left l-dagger-empty-lhs*)

lemma *k-finite-munion*:
$finite\ (dom\ f) \Longrightarrow finite(dom\ g) \Longrightarrow dom\ f \cap dom\ g = \{\} \Longrightarrow finite(dom(f \cup m\ g))$
by (*metis finite-Un l-munion-dom*)

lemma *l-munion-singleton-not-empty*:
$x \notin dom\ f \Longrightarrow f \cup m\ [x \mapsto y] \neq empty$
apply (*cases* $f = empty$)
apply (*metis l-munion-empty-lhs map-upd-nonempty*)
unfolding *munion-def*
apply *simp*
by (*metis dagger-def map-add-None*)

lemma *l-munion-empty-iff*:
$dom\ f \cap dom\ g = \{\} \Longrightarrow (f \cup m\ g = empty) \longleftrightarrow (f = empty \wedge g = empty)$
apply (*rule iffI*)
apply (*simp only*: *dom-eq-empty-conv*[*symmetric*] *l-munion-dom*)
apply (*metis Un-empty*)
by (*simp add*: *l-munion-empty-lhs l-munion-empty-rhs*)

lemma *l-munion-dom-ar-singleton-subsume*:
$x \notin dom\ f \Longrightarrow \{x\} \mathbin{-\!\!\lhd} (f \cup m\ [x \mapsto y]) = f$
apply (*subst fun-eq-iff*)

apply (*rule allI*)
unfolding *dom-antirestr-def*
by (*auto simp*: *l-munion-apply*)

lemma *l-munion-upd*: *dom f* ∩ *dom* [*x* ↦ *y*] = {} ⟹ *f* ∪*m* [*x* ↦ *y*] = *f*(*x* ↦*y*)
unfolding *munion-def*
apply *simp*
by (*metis dagger-def map-add-empty map-add-upd*)

lemma *munion-notemp-dagger*: *dom f* ∩ *dom g* = {} ⟹ *f* ∪*m g*≠*empty* ⟹ *f* † *g* ≠ *empty*
by (*metis munion-def*)

lemma *dagger-notemp-munion*: *dom f* ∩ *dom g* = {} ⟹ *f* † *g*≠*empty* ⟹ *f* ∪*m g* ≠ *empty*
by (*metis munion-def*)

lemma *munion-notempty-left*: *dom f* ∩ *dom g* = {} ⟹ *f* ≠ *empty* ⟹ *f* ∪*m g* ≠ *empty*
by (*metis dagger-notemp-munion dagger-notemptyL*)

lemma *munion-notempty-right*: *dom f* ∩ *dom g* = {} ⟹ *g* ≠ *empty* ⟹ *f* ∪*m g* ≠ *empty*
by (*metis dagger-notemp-munion dagger-notemptyR*)

lemma *unionm-in-dom-left*: *x* ∈ *dom* (*f* ∪*m g*) ⟹ (*dom f* ∩ *dom g*) = {} ⟹ *x* ∉ *dom g* ⟹ *x* ∈ *dom f*
by (*simp add*: *l-munion-dom*)

lemma *unionm-in-dom-right*: *x* ∈ *dom* (*f* ∪*m g*) ⟹ (*dom f* ∩ *dom g*) = {} ⟹ *x* ∉ *dom f* ⟹ *x* ∈ *dom g*
by (*simp add*: *l-munion-dom*)

lemmas *munion-simps* = *k-munion-map-upd-wd l-munion-apply l-munion-dom b-dagger-munion l-munion-subsume l-munion-subsumeG l-munion-dom-ar-assoc l-munion-empty-rhs l-munion-empty-lhs k-finite-munion l-munion-upd munion-notemp-dagger dagger-notemp-munion munion-notempty-left munion-notempty-right*

lemmas *vdm-simps* = *restr-simps antirestr-simps dagger-simps upd-simps munion-simps*

D.3.0.7 Map finiteness weakening lemmas [EXPERT]

— Need to have the lemma options, otherwise it fails somehow
lemma *finite-map-upd-induct* [*case-names empty insert*, *induct set*: *finite*]:
assumes *fin*: *finite* (*dom f*)
and *empty*: *P Map.empty*
and *insert*: ⋀*e r f*. *finite* (*dom f*) ⟹ *e* ∉ *dom f* ⟹ *P f* ⟹ *P* (*f*(*e* ↦ *r*))
shows *P f* **using** *fin*
proof (*induct dom f arbitrary*: *f rule*:*finite-induct*) — arbitrary statement is a must in here, otherwise cannot prove it
case *goal1* **then have** *dom f* = {} **by** *simp* — need to reverse to apply rules
then have *f* = *Map.empty* **by** *simp*
thus *?case* **by** (*simp add*: *empty*)
next
case *goal2*
— Show that update of the domain means an update of the map

```
  assume domF: insert x F = dom f then have domFr: dom f = insert x F by simp
  then obtain f0 where f0Def: f0 = f |` F by simp
  with domF have domF0: F = dom f0 by auto
  with goal2 have finite (dom f0) and x ∉ dom f0 and P f0 by simp-all
  then have PFUpd: P (f0(x ↦ the (f x))) by (rule insert)
  from domFr f0Def have f = f0(x ↦ the (f x)) by (auto intro: l-insertUpdSpec-aux)
  with PFUpd show ?case by simp
qed

lemma finiteRan: finite (dom f) ⟹ finite (ran f)
proof (induct rule:finite-map-upd-induct)
  case goal1 thus ?case by simp
next
  case goal2 then have ranIns: ran (f(e ↦ r)) = insert r (ran f) by auto
  assume finite (ran f) then have finite (insert r (ran f)) by (intro finite.insertI)
  thus ?case apply (subst ranIns)
 by simp
qed



lemma l-dom-r-finite: finite (dom f) ⟹ finite (dom ( S ◁ f))
apply (rule-tac B=dom f in  finite-subset)
apply (simp add: l-dom-r-dom-subseteq)
apply assumption
done

lemma dagger-finite: finite (dom f) ⟹ finite (dom g) ⟹ finite (dom (f † g))
    by (metis dagger-def dom-map-add finite-Un)

lemma finite-singleton: finite (dom [a ↦ b])
   by (metis dom-eq-singleton-conv finite.emptyI finite-insert)



lemma not-in-dom-ar: finite (dom f) ⟹ s ∩ dom f = {} ⟹ dom (s -◁ f) = dom f
apply (induct rule: finite-map-upd-induct)
apply (unfold dom-antirestr-def) apply simp
by (metis IntI domIff empty-iff)


lemma not-in-dom-ar-2: finite (dom f) ⟹ s ∩ dom f = {} ⟹ dom (s -◁ f) = dom f
apply (subst set-eq-subset)
apply (rule conjI)
apply (rule-tac[!] subsetI)
apply (metis l-dom-ar-not-in-dom)
by (metis l-dom-ar-nothing)



lemma l-dom-ar-commute-quickspec:
  S -◁ (T -◁ f) = T -◁ (S -◁ f)
by (metis l-dom-ar-accum sup-commute)

lemma l-dom-ar-same-subsume-quickspec:
  S -◁ (S -◁ f) = S -◁ f
```

by (*metis l-dom-ar-accum sup-idem*)

end

Appendix E

Heap lemmas and proofs (Leo)

theory *HEAP0Lemmas*
imports *HEAP0*
begin

E.1 HEAP0 Isabelle (automation) lemmas

E.1.1 locs_of weakening lemmas [EXPERT]

lemma *b-locs-of-as-set-interval*:
$nat1\ n \Longrightarrow locs\text{-}of\ l\ n = \{l..<l+n\}$
unfolding *locs-of-def*
by (*metis Collect-conj-eq atLeastLessThan-def atLeast-def lessThan-def*)

lemma *b-locs-of-finite*:
$nat1\ n \Longrightarrow finite(locs\text{-}of\ i\ n)$
by (*metis finite-atLeastLessThan b-locs-of-as-set-interval*)

lemma *b-locs-of-non-empty*:
$nat1\ n \Longrightarrow locs\text{-}of\ l\ n \neq \{\}$
unfolding *locs-of-def*
by (*metis* (*lifting*) *Collect-empty-eq le-add1 nat1-def nat-add-left-cancel-less*)

lemma *l-locs-of-card*:
$nat1\ n \Longrightarrow card(locs\text{-}of\ l\ n) = n$
by (*metis add-diff-cancel-left' b-locs-of-as-set-interval card-atLeastLessThan*)

end

theory *HEAP0Proofs*
imports *HEAP0 HEAP0Lemmas*
begin

E.2 Feasibility proof obligations for HEAP level 0

context *level0-new*

begin

theorem
locale0-new-FSB: *PO-new0-feasibility*
unfolding *PO-new0-feasibility-def*
by (*metis F0-inv-defs*
finite-Diff
l0-invariant-def
new0-post-def
new0-postcondition-def
new0-pre-def
l0-new0-precondition-def)

end

context *level0-dispose*
begin

theorem
locale0-dispose-FSB: *PO-dispose0-feasibility*
unfolding *PO-dispose0-feasibility-def dispose0-postcondition-def dispose0-post-defs*
by (*metis* (*full-types*) *F0-inv-defs finite-M-bounded-by-nat finite-Un l0-input-notempty-def l0-invariant-def*)

end

end

theory *HEAP0SanityProofs*
imports *HEAP0Sanity HEAP0Proofs*
begin

E.3 Proof of some properties of interest

E.3.1 Invariant

lemma *l-F0-inv-example*: *F0-ex-inv F0-ex*
unfolding *F0-ex-inv-defs* **by** *auto*

lemma *l-F0-inv-counter-example*: ¬ *F0-ex-inv UNIV*
unfolding *F0-ex-inv-defs* **by** *auto*

E.3.2 Operations

lemma *new0-post-shrinks-f*:
PO-new0-post-shrinks-f
unfolding *PO-new0-post-shrinks-f-def new0-post-defs*
by (*smt Diff-subset mem-Collect-eq nat1-def set-diff-eq set-mp subset-iff-psubset-eq*)

context *level0-new*
begin

lemma *new0-postcondition-shrinks-f*:
PO-new0-postcondition-shrinks-f
by (*smt PO-new0-post-shrinks-f-def PO-new0-postcondition-shrinks-f-def new0-post-shrinks-f new0-postcondition-def*)

end

thm *card-Diff-subset*[*of locs-of r n f*]

lemma *new0-post-shrinks-f-exactly*:
PO-new0-post-shrinks-f-exactly
unfolding *PO-new0-post-shrinks-f-exactly-def new0-post-def is-block-def*
apply *safe*
by (*simp add*: *card-Diff-subset b-locs-of-finite b-locs-of-as-set-interval*)

context *level0-dispose*
begin

lemma *dispose0-postcondition-extends-f*:
PO-dispose0-postcondition-extends-f
unfolding *PO-dispose0-postcondition-extends-f-def*
dispose0-postcondition-def
dispose0-post-def
by (*metis Un-commute b-locs-of-non-empty*
dispose0-pre-def inf-sup-absorb
inf-sup-ord(*3*) *l0-dispose0-precondition-def*
l0-input-notempty-def less-le)

lemma
dispose0-postcondition f′ $\Longrightarrow$ *f0* $\subset$ *f′*
unfolding *dispose0-postcondition-def dispose0-post-def*
by (*metis b-locs-of-non-empty dispose0-pre-def*
l0-dispose0-precondition-def inf-absorb2
inf-commute inf-sup-ord(*4*) *l0-input-notempty-def*
le-iff-sup less-le sup.left-idem)

thm *card-Un-disjoint*[*of f0 locs-of d0 s0*]

lemma *dispose0-postcondition-extends-f-exactly*:
PO-dispose0-postcondition-extends-f-exactly
unfolding *PO-dispose0-postcondition-extends-f-exactly-def*
dispose0-postcondition-def dispose0-post-def F0-inv-def
by (*metis Int-commute add-0-iff card-Un-Int card-empty*
dispose0-pre-def finite-Un l0-dispose0-precondition-def
l0-input-notempty-def l-locs-of-card)

lemma
dispose0-postcondition f′ $\Longrightarrow$ *card f′* = *card f0* + *s0*
unfolding *dispose0-postcondition-def dispose0-post-def F0-inv-def*

by (*metis card.union-inter card-empty comm-monoid-add-class.add.left-neutral dispose0-pre-def finite-Un inf-commute l0-dispose0-precondition-def l0-input-notempty-def l-locs-of-card nat-add-commute*)

end

lemma *PO-new0-dispose-0-identity*
unfolding *PO-new0-dispose-0-identity-def new0-post-def dispose0-post-def F0-inv-def*
apply (*safe*)
apply (*metis is-block-def set-mp*)
apply (*metis finite-Diff*)
by (*metis b-locs-of-finite is-block-def*)

lemma *new0-dispose-0-identity*:
PO-new0-dispose-0-identity
by (*metis PO-new0-dispose-0-identity-def Un-Diff-cancel dispose0-post-def is-block-def new0-post-def sup-absorb1 sup-commute*)

end

theory *HEAP1Lemmas*
imports *HEAP1 HEAP0Lemmas*
begin

This theory provides various lemmas for breaking the problem into managelable chunks

E.4 General Lemmas

These lemmas are used in the context of NEW1 FSB locale proofs. Prefixes determine the intent (our whys?) as given by the expert. Depending on context, some intents could have more than one prefix or even change prefix (as determined by the expert). These "tags" should serve as clues for strategy languages and learning mechanisms to infer new (useful) lemmas or indeed strategies (proof patterns).

Prefixes: "k_" = weakening goal (barkward reasonsing) "f_" = deduction from asm (forward reasonsing) "b_" = type/concept bridges "l_" = expert lemmas

E.4.0.1 nat1_map weakening lemmas [EXPERT]

lemma *f-nat1-map-nat1-elem*:
nat1-map f $\Longrightarrow$ *x* $\in$ *dom f* $\Longrightarrow$ *0* $<$ (*the*(*f x*))
by (*metis nat1-def nat1-map-def*)

lemma *f-nat1-map-extends-map-le*:
g $\subseteq_m$ *f* $\Longrightarrow$ *nat1-map f* $\Longrightarrow$ *nat1-map g*
apply (*frule map-le-implies-dom-le*)
unfolding *map-le-def nat1-map-def*
apply (*intro allI impI*)

apply (*drule bspec*, *assumption*)
apply (*drule spec*, *drule mp*)
apply *simp-all*
by (*drule subsetD*, *assumption*)

lemma *k-nat1-map-dom-ar*:
nat1-map f $\Longrightarrow$ *nat1-map* (*S* -⩤ *f*)
by (*metis nat1-map-def f-in-dom-ar-subsume f-in-dom-ar-the-subsume*)

lemma *k-nat1-map-dom-ar-specific*:
nat1-map f $\Longrightarrow$ *nat1-map* ({*r*} -⩤ *f*)
by (*metis k-nat1-map-dom-ar*)

lemma *l-nat1-map-dagger*: *nat1-map f* $\Longrightarrow$ *nat1-map g* $\Longrightarrow$ *nat1-map*(*f* † *g*)
unfolding *nat1-map-def*
apply (*intro allI impI*)
apply (*simp add*: *l-dagger-dom l-dagger-apply*)
by *metis*

lemma *l-nat1-map-munion*: *nat1-map f* $\Longrightarrow$ *nat1-map g* $\Longrightarrow$ *dom f* ∩ *dom g* = {} $\Longrightarrow$ *nat1-map*(*f* ∪*m g*)
unfolding *nat1-map-def*
apply (*intro allI impI*)
apply (*simp add*: *l-munion-dom l-munion-apply*)
by *metis*

lemma *l-nat1-map-singleton*: *nat1 y* $\Longrightarrow$ *nat1-map*([*x* ↦ *y*])

by (*metis fun-upd-triv map-add-empty map-add-upd map-le-map-add nat1-map-def f-nat1-map-extends-map-le the.simps*)

lemma *l-nat1-map-empty*: *nat1-map empty*
by (*metis dom-empty empty-iff nat1-map-def*)

E.4.0.2 locs_of weakening lemmas [EXPERT]

These lemmas were useful in the Z/EVES development and now here. At first we had difficulties with the style of declaration as intro/elim/dest rules. I tried to keep them as iff is possible.

lemma *l-locs-of-Locs-of-iff*:
l ∈ *dom f* $\Longrightarrow$ *Locs-of f l* = *locs-of l* (*the* (*f l*))
unfolding *Locs-of-def*
by *simp*

lemma *k-locs-of-arithIff*:
nat1 n $\Longrightarrow$ *nat1 m* $\Longrightarrow$ (*locs-of a n* ∩ *locs-of b m* = {}) = (*a*+*n* ≤ *b* ∨ *b*+*m* ≤ *a*)
unfolding *locs-of-def*
apply *simp*

apply (*rule iffI*)
find-theorems - ∩ - = {}
apply (*erule equalityE*)
apply (*simp-all add*: *disjoint-iff-not-equal*)
apply (*metis* (*full-types*) *add-0-iff le-add1 le-neq-implies-less nat-le-linear not-le*)
by (*metis le-trans not-less*)

lemma *k-locs-of-dom-ar-subset*:
nat1-map f ⟹ *x* ∈ *dom* (*S* -⊲ *f*) ⟹ *locs-of x* (*the*((*S* -⊲ *f*) *x*)) ⊆ *locs-of x* (*the*(*f x*))
apply (*frule k-nat1-map-dom-ar*[*of* - *S*])
apply (*frule f-nat1-map-nat1-elem*[*of S* -⊲ *f* -],*assumption*)

apply (*rule subsetI*)
by (*metis f-in-dom-ar-apply-subsume*)

lemma *k-Locs-of-arithIff*:
nat1-map f ⟹ *l* ∈ *dom f* ⟹ *k* ∈ *dom f* ⟹ (*Locs-of f l* ∩ *Locs-of f k* = {}) = (*l*+*the*(*f l*) ≤ *k* ∨ *k*+*the*(*f k*) ≤ *l*)
unfolding *Locs-of-def*
by (*simp add*: *f-nat1-map-nat1-elem k-locs-of-arithIff*)

E.4.0.3 locs weakening lemmas [EXPERT]

lemma *k-in-locs-iff*: *nat1-map f* ⟹ (*x* ∈ *locs f*) = (∃·*y* ∈ *dom f* . *x* ∈ *locs-of y* (*the*(*f y*)))
unfolding *locs-def*
by (*metis* (*mono-tags*) *UN-iff*)

lemma *l-locs-of-within-locs*:
nat1-map f ⟹ *x* ∈ *dom f* ⟹ *locs-of x* (*the*(*f x*)) ⊆ *locs f*
by (*metis k-in-locs-iff subsetI*)

lemma *k-inter-locs-iff*: *nat1 s* ⟹ *nat1-map f* ⟹ (*locs-of x s* ∩ *locs f* = {}) = (∀· *y* ∈ *dom f* . *locs-of x s* ∩ *locs-of y* (*the*(*f y*)) = {})
unfolding *locs-def*
by (*smt UNION-empty-conv*(*1*) *inf-SUP*)

lemma *l-locs-subset*:
nat1-map f ⟹ *g* ⊆$_m$ *f* ⟹ *locs g* ⊆ *locs f*
apply (*frule f-nat1-map-extends-map-le*, *assumption*)
apply (*rule subsetI*)
unfolding *locs-def*
apply (*simp*)
apply (*erule bexE*)
apply (*frule map-le-implies-dom-le*)
unfolding *map-le-def*
apply (*drule bspec*, *assumption*)
thm *in-mono set-rev-mp set-mp*
by (*metis set-mp*)

lemma *l-locs-dom-ar-iff*:
nat1-map f $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ $r \in$ *dom f* $\Longrightarrow$ *locs*($\{r\}$ -$\lhd$ *f*) = *locs f* - *locs-of r* (*the*(*f r*))
apply (*rule equalityI*)
apply (*rule-tac* [*1*-] *subsetI*)
apply (*frule-tac* [*1*-] *k-nat1-map-dom-ar*[*of* - $\{r\}$])
apply (*simp-all add*: *k-in-locs-iff*)
defer
apply (*elim conjE*)
defer
apply (*intro conjI*)
apply (*metis f-in-dom-ar-subsume f-in-dom-ar-the-subsume*)
apply (*erule-tac* [*1*-] *bexE*)
defer
apply (*rule-tac x*=*y* **in** *bexI*)
apply (*metis f-in-dom-ar-apply-not-elem singleton-iff*)
apply (*metis l-dom-dom-ar member-remove remove-def*)
apply (*frule f-in-dom-ar-subsume*)
apply (*frule f-in-dom-ar-the-subsume*)
unfolding *Disjoint-def disjoint-def*
apply (*simp add*: *l-locs-of-Locs-of-iff*)
by (*metis disjoint-iff-not-equal f-in-dom-ar-notelem*)

lemma *l-locs-dom-ar-general-iff*:
nat1-map f $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *S* $\subseteq$ *dom f* $\Longrightarrow$ *locs*(*S* -$\lhd$ *f*) = *locs f* - ($\bigcup r{\in}S$. *locs-of r* (*the*(*f r*)))
apply (*rule equalityI*)
apply (*rule-tac* [*1*-] *subsetI*)
apply (*frule-tac* [*1*-] *k-nat1-map-dom-ar*[*of* - *S*])
apply (*simp-all add*: *k-in-locs-iff*)
defer
apply (*elim conjE*)
defer
apply (*intro conjI*)
apply (*metis f-in-dom-ar-subsume f-in-dom-ar-the-subsume*)
apply (*erule-tac* [*1*-] *bexE*)
apply (*rule ballI*)
apply (*metis Disjoint-def disjoint-def disjoint-iff-not-equal f-in-dom-ar-apply-subsume l-dom-ar-notin-dom-or l-locs-of-Locs-of-iff set-rev-mp*)
apply (*cases S* = {})
apply (*simp add*: *l-dom-ar-none*)
apply *metis*

find-theorems *simp*:- $\neq$ {}
apply (*simp add*: *nonempty-iff*)
apply (*elim exE conjE*)
by (*metis f-in-dom-ar-apply-not-elem l-in-dom-ar*)

lemma *l-locs-empty-iff*:
locs empty = {}
apply (*rule equalityI*)
apply (*rule-tac* [*1*-] *subsetI*)
apply *simp-all*
apply (*subgoal-tac nat1-map empty*)
apply (*simp add*: *locs-def*)
by (*rule l-nat1-map-empty*)

lemma *l-locs-singleton-iff*:
  nat1 y $\Longrightarrow$ *locs* $[x \mapsto y]$ = *locs-of x y*
unfolding *locs-def locs-of-def nat1-map-def*
by *simp*

lemma *f-dom-locs-of*: *nat1-map f* $\Longrightarrow$ $(x \in dom\ f) \Longrightarrow (x \in$ *locs-of x* $(the\ (f\ x)))$
unfolding *locs-of-def*
by (*simp add*: *f-nat1-map-nat1-elem*)

lemma *f-in-dom-locs*: *nat1-map f* $\Longrightarrow x \in dom\ f \Longrightarrow x \in locs\ f$
apply (*simp add*: *k-in-locs-iff*)
apply (*rule bexI*)
by (*simp-all add*: *f-dom-locs-of*)

lemma *l-locs-munion-iff*:
    nat1-map f $\Longrightarrow$ *nat1-map g* $\Longrightarrow$ *dom f* $\cap$ *dom g* = $\{\}$ $\Longrightarrow$ *locs*$(f \cup m\ g)$ = *locs f* $\cup$ *locs g*
apply (*rule equalityI*)
apply (*rule-tac* [*1*-] *subsetI*) — Little trick to cover all goals
apply *simp-all*
apply (*rule disjCI*) — Keep the contrapositive information; it's useful later
defer
apply (*erule disjE*)
apply (*simp-all add*: *k-in-locs-iff l-nat1-map-munion l-munion-dom l-munion-apply*)
apply (*erule-tac* [*1*-] *bexE*)
apply (*rule-tac* [*1-2*] *x*=*y* **in** *bexI*)
apply (*simp-all*)
apply (*metis disjoint-iff-not-equal*)

  thm *all-not-in-conv*
apply (*erule disjE*)
apply (*rule-tac x*=*y* **in** *bexI*)
apply (*metis* (*full-types*))
apply *assumption*
by (*metis* (*full-types*))

lemma *l-locs-dagger-union-subset*:
  nat1-map f $\Longrightarrow$ *nat1-map g* $\Longrightarrow$ *locs*$(f \dagger g) \subseteq$ *locs f* $\cup$ *locs g*
apply (*rule subsetI*)
apply *simp*
apply (*rule disjCI*)
apply (*simp-all add*: *k-in-locs-iff l-nat1-map-dagger l-dagger-dom l-dagger-apply*)
apply (*erule bexE*)
apply *simp*
apply (*erule disjE*)
apply (*metis* (*full-types*))
by (*metis* (*full-types*))

lemma *l-locs-dagger-iff*:
nat1-map f $\Longrightarrow$ *nat1-map g* $\Longrightarrow$ $(\forall \cdot\ x \in dom\ f \cap dom\ g\ .\ the(f\ x) \leq the(g\ x)) \Longrightarrow locs(f \dagger g) = locs\ f \cup locs\ g$
apply (*rule equalityI*)
apply (*simp add*: *l-locs-dagger-union-subset*)
apply (*rule subsetI*)
apply *simp*
apply (*erule disjE*)
apply (*simp-all add*: *k-in-locs-iff l-nat1-map-dagger l-dagger-dom l-dagger-apply*)
apply (*erule-tac* [*1-*] *bexE*)
apply (*rule-tac* [*1-*] *x=y* **in** *bexI*)
apply (*simp-all*)
apply (*rule impI*)
apply (*simp add*: *b-locs-of-as-set-interval f-nat1-map-nat1-elem*)
apply (*erule conjE*)
apply (*erule-tac x=y* **in** *ballE*)
by *simp-all*

E.4.0.4 min_loc lemmas [EXPERT]

lemma *k-min-loc-munion*:
finite (*dom f*) $\Longrightarrow$ *finite* (*dom g*) $\Longrightarrow$
g $\neq$ *empty* $\Longrightarrow$ *dom f* $\cap$ *dom g* = {} $\Longrightarrow$
min-loc(*f* $\cup m$ *g*) = (*if f* = *empty then min-loc g else min* (*min-loc f*) (*min-loc g*))
unfolding *min-loc-def munion-def*
by (*simp add*: *l-dagger-not-empty l-dagger-dom Min-Un*)

lemma *l-min-loc-singleton*:
min-loc [*d* $\mapsto$ *s*] = *d*
unfolding *min-loc-def*
by *simp*
— by (metis dom_empty finite.emptyI inf_bot_left k_min_loc_munion_singleton l_munion_empty_lhs)
= Overkill!

lemma *k-min-loc-munion-singleton*:
finite (*dom f*) $\Longrightarrow$
dom f $\cap$ *dom* [*d* $\mapsto$ *s*] = {} $\Longrightarrow$
min-loc(*f* $\cup m$ [*d* $\mapsto$ *s*]) = (*if f* = *empty then d else min* (*Min* (*dom f*)) *d*)
apply (*simp add*: *k-min-loc-munion l-min-loc-singleton*)
by (*metis min-loc-def*)

E.4.0.5 sum_size lemmas [EXPERT]

lemma *l-sum-size-munion*:
finite (*dom f*) $\Longrightarrow$ *finite* (*dom g*) $\Longrightarrow$
g $\neq$ *empty* $\Longrightarrow$ *dom f* $\cap$ *dom g* = {} $\Longrightarrow$
sum-size(*f* $\cup m$ *g*) = (*if f* = *empty then sum-size g else* (*sum-size f*) + (*sum-size g*))
unfolding *sum-size-def munion-def*
apply (*simp add*: *l-dagger-not-empty l-dagger-empty-lhs l-dagger-dom l-dagger-apply*)
apply (*rule impI*)
find-theorems $(\sum$ - $\in$ - . -) = (($\sum$ - $\in$ - . -) + ($\sum$ - $\in$ - . -))
thm *setsum.F-Un-neutral*[*of dom f dom g* (λ *x* . *the* (*if x* $\in$ *dom g then g x else f x*)),*simplified*]
thm *setsum-Un-disjoint*[*of dom f dom g* (λ *x* . *the* (*if x* $\in$ *dom g then g x else f x*)),*simplified*]
apply (*simp add*: *setsum-Un-disjoint*)
apply (*rule setsum-cong*,*simp*)
by (*metis* (*full-types*) *disjoint-iff-not-equal*)

lemma *l-sum-size-singleton*:
sum-size $[d \mapsto s] = s$
unfolding *sum-size-def*
by *simp*

lemma *l-sum-size-munion-singleton*:
finite $(dom\ f) \Longrightarrow$
$dom\ f \cap dom\ [d \mapsto s] = \{\} \Longrightarrow$
$sum\text{-}size(f \cup m\ [d \mapsto s]) = (if\ f = empty\ then\ s\ else\ sum\text{-}size\ f + s)$
by (*simp add*: *l-sum-size-munion l-sum-size-singleton*)

E.4.0.6 Other (less useful) lemmas [EXPERT]

lemma *l-disjoint-comm*:
$(disjoint\ A\ B) = (disjoint\ B\ A)$
by (*metis disjoint-def inf-commute*)

lemma *f-F1-inv-disjoint*:
F1-inv f $\Longrightarrow$ *Disjoint f*
by (*metis F1-inv-def*)

lemma *f-F1-inv-nat1-map*:
F1-inv f $\Longrightarrow$ *nat1-map f*
by (*metis F1-inv-def*)

lemma *f-F1-inv-sep*:
F1-inv f $\Longrightarrow$ *sep f*
by (*metis F1-inv-def*)

lemma *f-F1-inv-finite*:
F1-inv f $\Longrightarrow$ $finite(dom\ f)$
by (*metis F1-inv-def*)

E.5 Goal-oriented - invariant update

E.5.0.7 Lemmas for invariant sub parts over known operators

This is a great example of repeated patterns.

lemma *l-sep-singleton*: *nat1 y* $\Longrightarrow$ $sep([x \mapsto y])$
unfolding *sep-def*
by *simp*

definition
sep0 :: *F1* $\Rightarrow$ *F1* $\Rightarrow$ *bool*
where
sep0 f g $\equiv (\forall \cdot\ l \in dom\ f\ .\ l + the(f\ l) \notin dom\ g)$

lemma *sep0 f f* = *sep f*
unfolding *sep0-def sep-def*
by *simp*

lemma *l-sep-singleton-upd*:
nat1-map f $\Longrightarrow$ $x \notin dom\ f \Longrightarrow x+y \notin dom\ f \Longrightarrow$ *nat1 y* $\Longrightarrow$ *sep f* $\Longrightarrow$
sep0 f $[x \mapsto y] \Longrightarrow sep(f \cup m\ [x \mapsto y])$

unfolding *sep-def sep0-def*
apply (*rule ballI*)
apply (*simp add*: *l-munion-dom l-munion-apply*)
apply (*erule disjE*)
by (*simp-all*)

lemma *l-sep-munion*:
  $dom\ f \cap dom\ g = \{\} \Longrightarrow sep\ f \Longrightarrow sep\ g \Longrightarrow sep0\ f\ g \Longrightarrow sep0\ g\ f \Longrightarrow sep(f \cup m\ g)$
unfolding *sep-def sep0-def*
by (*auto simp*: *l-munion-dom l-munion-apply*)

lemma *nat1-map f* $\Longrightarrow$ $x \notin dom\ f \Longrightarrow$ *nat1 y* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$
    $(\forall \cdot\ c \in dom\ f\ .\ x+y \leq c \vee c+the(f\ c) \leq x) \Longrightarrow Disjoint(f \cup m\ [x \mapsto y])$
unfolding *Disjoint-def*
apply (*simp add*: *l-locs-of-Locs-of-iff*)
apply (*intro ballI impI*)
apply (*simp add*: *l-munion-dom l-munion-apply*)
apply (*intro conjI impI*)
apply (*simp-all add*: *l-disjoint-comm*)
unfolding *disjoint-def*
find-theorems *locs-of - - ∩ - = {}*
by (*simp-all add*: *k-locs-of-arithIff f-nat1-map-nat1-elem*)

thm *k-locs-of-arithIff*[*of y the*(*f c*) *x c*,*symmetric*]

lemma *l-disjoint-singleton-upd*:
  nat1-map f $\Longrightarrow$ $x \notin dom\ f \Longrightarrow$ *nat1 y* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$
    disjoint (*locs-of x y*) (*locs f*) $\Longrightarrow Disjoint(f \cup m\ [x \mapsto y])$
unfolding *Disjoint-def*
apply (*simp add*: *l-locs-of-Locs-of-iff*)
apply (*intro ballI impI*)
apply (*simp add*: *l-munion-dom l-munion-apply*)
apply (*intro conjI impI*)
apply (*simp-all*)
unfolding *disjoint-def*
find-theorems *locs -*
find-theorems *locs-of - - ∩ - = {}*
apply (*metis k-inter-locs-iff nat1-def*)
by (*metis inf-commute k-inter-locs-iff nat1-def*)

lemma *l-disjoint-singleton*: $Disjoint([x \mapsto y])$
unfolding *Disjoint-def*
by *simp*

lemma *l-disjoint-munion*:
  nat1-map f $\Longrightarrow$ *nat1-map g* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *Disjoint g* $\Longrightarrow$
    $dom\ f \cap dom\ g = \{\} \Longrightarrow$ *disjoint* (*locs f*) (*locs g*) $\Longrightarrow$ *sep0 f g* $\Longrightarrow$ *sep0 g f* $\Longrightarrow$ *Disjoint* (*f* $\cup m$ *g*)
unfolding *Disjoint-def*
apply (*intro impI ballI*)
apply (*simp add*: *l-locs-of-Locs-of-iff l-munion-apply l-munion-dom*)
apply (*intro impI conjI*)
apply *simp-all*

apply (*simp-all add*: *l-locs-of-Locs-of-iff*[*symmetric*])
apply (*fold Disjoint-def*)
apply (*simp-all add*: *l-locs-of-Locs-of-iff*)
unfolding *disjoint-def*
find-theorems *name*:*arith name*:*loc*
apply (*frule-tac* [*1*-] *f-in-dom-locs*[*of f*],*simp-all*)
apply (*frule-tac* [*1*-] *f-in-dom-locs*[*of g*],*simp-all*)
find-theorems *name*:*disjoint name*:*iff*
find-theorems *locs* -
apply (*simp-all add*: *k-in-locs-iff*)
apply (*erule-tac* [*1*-] *bexE*)+
apply (*simp-all add*: *k-locs-of-arithIff f-nat1-map-nat1-elem*)
unfolding *sep0-def*
apply (*erule-tac x*=*b* **in** *ballE*,*simp-all*)
apply (*erule-tac x*=*a* **in** *ballE*,*simp-all*)
oops

lemma *l-nat1-map-singleton-upd*: *nat1 y* $\Longrightarrow$ *x* $\notin$ *dom f* $\Longrightarrow$ *nat1-map f* $\Longrightarrow$ *nat1-map*(*f* $\cup m$ [*x* $\mapsto$ *y*])
unfolding *nat1-map-def*
by (*simp add*: *l-munion-dom l-munion-apply*)

lemma *l-finite-singleton-upd*:
nat1 y $\Longrightarrow$ *x* $\notin$ *dom f* $\Longrightarrow$ *finite*(*dom f*) $\Longrightarrow$ *finite*(*dom*(*f* $\cup m$ [*x* $\mapsto$ *y*]))
by (*simp add*: *l-munion-dom*)

E.5.0.8 NEW1 update - equal case

Most lemmas are marked as weakening rules. That's because they used by the top-level goals for the proof obligations. In other scenarios, they could be used a deduction (FD) rules as well.

lemma *k-Disjoint-dom-ar*:
Disjoint f $\Longrightarrow$ *Disjoint* (*S* -$\lhd$ *f*)
by (*smt Disjoint-def Locs-of-def domIff dom-antirestr-def*)

lemma *k-sep-dom-ar*:
sep f $\Longrightarrow$ *sep* (*S* -$\lhd$ *f*)
by (*metis* (*full-types*) *f-in-dom-ar-subsume f-in-dom-ar-the-subsume sep-def*)

lemma *k-finite-dom-ar*:
finite (*dom f*) $\Longrightarrow$ *finite* (*dom* (*S* -$\lhd$ *f*))
by (*metis finite-subset f-in-dom-ar-subsume subsetI*)

lemma *k-F1-inv-dom-ar*:
F1-inv f $\Longrightarrow$ *F1-inv*(*S* -$\lhd$ *f*)
by (*metis F1-inv-def k-Disjoint-dom-ar k-finite-dom-ar k-nat1-map-dom-ar k-sep-dom-ar*)

E.5.0.9 NEW1 update - greater than case

In this final subsection, we get to the actual lemmas used by top-level goals. These lemmas were first defined in terms of $f \dagger g$, which later turned into $f \cup g$.

The proof strategy here is the same for each of the four parts of the invariant, providing we expose a key fact about the specific (greater than update) case: the updated value cannot be in dom f. This is crucial for the $(f \cup m\ g)$ operation to be well-defined.

A more specific lemma, useful only for the Disjoint invariant, is proved. It shows that the locations of the update are within the locations prior to the update, as expected. That is, we lift/bridge the update locations from the given value (r+s) to original (r).

lemma *l-disjoint-mapupd-keep-sep*:
nat1-map f $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *r* $\in$ *dom f* $\Longrightarrow$ *nat1 s* $\Longrightarrow$ *the*(*f r*) $>$ *s* $\Longrightarrow$ (*r*+*s*) $\notin$ *dom f*
unfolding *Disjoint-def*
apply (*erule-tac x*=*r* **in** *ballE*)
apply (*erule-tac x*=(*r*+*s*) **in** *ballE*)
apply (*erule impE*)
apply (*simp-all*)
apply (*rule notI*)
apply (*simp add*: *l-locs-of-Locs-of-iff*)
unfolding *disjoint-def*
by (*smt k-locs-of-arithIff nat1-map-def*)

lemma *k-new1-gr-dom-ar-dagger-aux2*:
nat1-map f $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *r* $\in$ *dom f* $\Longrightarrow$ *nat1 s* $\Longrightarrow$ *the*(*f r*) $>$ *s* $\Longrightarrow$ *r*+*s* $\notin$ *dom* ({*r*} -⊲ *f*)
by (*metis f-in-dom-ar-subsume l-disjoint-mapupd-keep-sep*)

lemma *k-new1-gr-dom-ar-dagger-aux*:
nat1-map f $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *r* $\in$ *dom f* $\Longrightarrow$ *nat1 s* $\Longrightarrow$ *the*(*f r*) $>$ *s* $\Longrightarrow$ *dom* ({*r*} -⊲ *f*) $\cap$ *dom* [*r* + *s* $\mapsto$ *the* (*f r*) - *s*] = {}
apply (*subst disjoint-iff-not-equal*)
by (*metis dom-eq-singleton-conv f-in-dom-ar-subsume l-disjoint-mapupd-keep-sep singletonE*)

lemma *b-new1-gr-upd-within-req-size*:
r $\in$ *dom f* $\Longrightarrow$ *the* (*f r*) $>$ *s* $\Longrightarrow$ *nat1-map f* $\Longrightarrow$
locs-of (*r*+*s*) (*the* (*f r*) - *s*) $\subseteq$ *locs-of r* (*the*(*f r*))
by (*simp add*: *b-locs-of-as-set-interval*)

find-theorems {- ..< -} $\subseteq$ {- ..< -}
thm *b-locs-of-as-set-interval*[*of the*(*f r*) - *s r* + *s*]
b-locs-of-as-set-interval[*of the*(*f r*) *r*]
ivl-subset[*of r* + *s r* + *s* + *the*(*f r*) - *s r the* (*f r*)]

lemma *b-new1-gr-upd-psubset-req-size*:
nat1 s $\Longrightarrow$ *r* $\in$ *dom f* $\Longrightarrow$ *the* (*f r*) $>$ *s* $\Longrightarrow$ *nat1-map f* $\Longrightarrow$
locs-of (*r*+*s*) (*the* (*f r*) - *s*) $\subset$ *locs-of r* (*the*(*f r*))
apply (*rule psubsetI*)
apply (*simp add*: *b-new1-gr-upd-within-req-size*)
apply (*simp add*: *b-locs-of-as-set-interval*)
by (*metis add-0-iff add-lessD1 add-less-cancel-left atLeastLessThan-inj*(*1*) *not-less0*)

lemma *k-Disjoint-dom-ar-dagger*:
r ∈ *dom f* ⟹ *the* (*f r*) > *s* ⟹ *nat1-map f* ⟹ *Disjoint f* ⟹ *Disjoint* (({*r*} -◁ *f*) † [*r* + *s* ↦ *the* (*f r*) - *s*])
unfolding *Disjoint-def disjoint-def*
apply (*intro impI ballI*)+
apply (*simp add*: *l-locs-of-Locs-of-iff l-dagger-apply*)
apply (*intro impI conjI*)+
apply (*simp-all add*: *l-dagger-dom*)
prefer *3*
apply (*metis f-in-dom-ar-subsume f-in-dom-ar-the-subsume*)
apply (*simp-all add*: *f-in-dom-ar-the-subsume*)
apply (*erule-tac x*=*r* **in** *ballE*)
apply (*erule-tac x*=*b* **in** *ballE*)
apply (*frule-tac* [*1-4*] *f-in-dom-ar-notelem*)
apply (*frule-tac* [*1-4*] *f-in-dom-ar-subsume*)
apply (*simp-all*)
thm *b-new1-gr-upd-within-req-size*[*of r f s*]
f-nat1-map-nat1-elem[*of f r*]
b-locs-of-as-set-interval[*of the*(*f r*)]

apply (*simp-all add*: *b-new1-gr-upd-within-req-size f-nat1-map-nat1-elem b-locs-of-as-set-interval*)
apply (*metis add-lessD1*)
done

lemma *k-Disjoint-dom-ar-munion*:
r ∈ *dom f* ⟹ *the* (*f r*) > *s* ⟹ *nat1 s* ⟹ *nat1-map f* ⟹ *Disjoint f* ⟹ *Disjoint* (({*r*} -◁ *f*) ∪*m* [*r* + *s* ↦ *the* (*f r*) - *s*])
apply (*frule l-disjoint-mapupd-keep-sep*[*of f r s*])
apply (*assumption*)+
unfolding *munion-def*
apply (*simp add*: *k-Disjoint-dom-ar-dagger*)
by (*metis f-in-dom-ar-subsume*)

lemma *k-sep-dom-ar-dagger-aux2*:
nat1 s ⟹ {*r*} ∩ *dom* [*r* + *s* ↦ *the* (*f r*) - *s*] = {}
apply (*subst disjoint-iff-not-equal*)
by *auto*

lemma *k-sep-dom-ar-dagger*:
r ∈ *dom f* ⟹ *the* (*f r*) > *s* ⟹ *nat1 s* ⟹ *sep f* ⟹ *Disjoint f* ⟹ *sep* ({*r*} -◁ *f* † [*r* + *s* ↦ *the* (*f r*) - *s*])
apply (*insert k-sep-dom-ar-dagger-aux2*[*of s r f*])
apply (*simp add*: *l-dagger-dom-ar-assoc*)
apply (*rule k-sep-dom-ar*)
unfolding *sep-def*
apply (*intro ballI*)
apply (*simp add*: *l-dagger-apply l-dagger-dom*)
apply (*intro impI conjI*)

apply (*simp-all*)
apply (*erule-tac x=l* **in** *ballE*)
apply (*simp-all*)
unfolding *Disjoint-def disjoint-def*
by (*smt l-locs-of-Locs-of-iff k-locs-of-arithIff nat1-def*)

lemma *k-sep-dom-ar-munion*:
nat1-map f $\Longrightarrow$ *r* $\in$ *dom f* $\Longrightarrow$ *the* (*f r*) > *s* $\Longrightarrow$ *nat1 s* $\Longrightarrow$ *sep f* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *sep* ({*r*} -$\triangleleft$ *f* $\cup m$ [*r* + *s* $\mapsto$ *the* (*f r*) - *s*])
unfolding *munion-def*
apply (*simp add*: *k-sep-dom-ar-dagger*)
by (*metis l-disjoint-mapupd-keep-sep f-in-dom-ar-subsume nat1-def*)

lemma *k-nat1-map-dom-ar-dagger*:
nat1 s $\Longrightarrow$ *r* $\in$ *dom f* $\Longrightarrow$ *the* (*f r*) > *s* $\Longrightarrow$ *nat1-map f* $\Longrightarrow$ *nat1-map* ({*r*} -$\triangleleft$ *f* † [*r* + *s* $\mapsto$ *the* (*f r*) - *s*])
unfolding *nat1-map-def*
apply (*intro allI impI*)
apply (*simp add*: *l-dagger-dom l-dagger-apply*)
apply (*intro conjI impI*)+
apply (*simp*)
by (*metis f-in-dom-ar-subsume f-in-dom-ar-the-subsume*)

lemma *k-nat1-map-dom-ar-munion*:
nat1 s $\Longrightarrow$ *r* $\in$ *dom f* $\Longrightarrow$ *the* (*f r*) > *s* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *nat1-map f* $\Longrightarrow$ *nat1-map* ({*r*} -$\triangleleft$ *f* $\cup m$ [*r* + *s* $\mapsto$ *the* (*f r*) - *s*])
unfolding *munion-def*
apply (*simp add*: *k-nat1-map-dom-ar-dagger*)
by (*metis l-disjoint-mapupd-keep-sep f-in-dom-ar-subsume nat1-def*)

lemma *k-finite-dom-ar-dagger*:
r $\in$ *dom f* $\Longrightarrow$ *the* (*f r*) > *s* $\Longrightarrow$ *finite* (*dom f*) $\Longrightarrow$ *finite* (*dom*({*r*} -$\triangleleft$ *f* † [*r* + *s* $\mapsto$ *the* (*f r*) - *s*]))
by (*simp add*: *l-dagger-dom l-dagger-apply k-finite-dom-ar*)

lemma *k-finite-dom-ar-munion*:
r $\in$ *dom f* $\Longrightarrow$ *the* (*f r*) > *s* $\Longrightarrow$ *nat1 s* $\Longrightarrow$ *nat1-map f* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *finite* (*dom f*) $\Longrightarrow$ *finite* (*dom*({*r*} -$\triangleleft$ *f* $\cup m$ [*r* + *s* $\mapsto$ *the* (*f r*) - *s*]))
unfolding *munion-def*
apply (*simp add*: *k-finite-dom-ar-dagger*)
by (*metis l-disjoint-mapupd-keep-sep f-in-dom-ar-subsume nat1-def*)

lemma *k-finite-dom-ar-munion-ALT-PROOF*:
r+*s* $\notin$ *dom f* $\Longrightarrow$ *r* $\in$ *dom f* $\Longrightarrow$ *the* (*f r*) > *s* $\Longrightarrow$ *finite* (*dom f*) $\Longrightarrow$ *finite* (*dom*({*r*} -$\triangleleft$ *f* $\cup m$ [*r*

$+\ s \mapsto the\ (f\ r)$ - $s]]$)
thm *l-munion-dom*[*of* $\{r\}$ -⩤*f* $[r + s \mapsto the(f\ r)$ - $s]]$
apply (*insert l-munion-dom*[*of* $\{r\}$ -⩤*f* $[r + s \mapsto the(f\ r)$ - $s]]$)
apply (*insert f-dom-ar-subset-dom*[*of* $\{r\}$ f])
apply (*simp*)
by (*metis finite-Diff finite-insert l-dom-dom-ar f-in-dom-ar-subsume*)

lemma *k-F1-inv-dom-munion*:
$F1\text{-}inv\ f \Longrightarrow nat1\ s \Longrightarrow r \in dom\ f \Longrightarrow the(f\ r) > s \Longrightarrow F1\text{-}inv(\{r\}$ -⩤ $f \cup m\ [r + s \mapsto the\ (f\ r)$ - $s])$
by (*metis F1-inv-def k-Disjoint-dom-ar-munion k-finite-dom-ar-munion k-nat1-map-dom-ar-munion k-sep-dom-ar-munion*)

E.6 Goal-oriented - DISPOSE1 invariant update

E.6.0.10 DISPOSE1 update - equal case

lemma *l-min-loc-dom-r-iff*:
$S \lhd g \neq empty \Longrightarrow min\text{-}loc\ (S \lhd g) = Min\ (S \cap dom\ g)$
by (*metis min-loc-def l-dom-r-iff*)

lemma *k-Min-subset*:
$S \neq \{\} \Longrightarrow finite\ T \Longrightarrow S \subseteq T \Longrightarrow Min\ S \in T$
by (*metis Min-in finite-subset set-mp*)

lemma *k-min-loc-dom*:
$g \neq empty \Longrightarrow finite(dom\ g) \Longrightarrow dom\ g \subseteq dom\ f \Longrightarrow min\text{-}loc\ g \in dom\ f$
unfolding *min-loc-def*
by (*metis Min-in dom-eq-empty-conv set-mp*)

lemma *k-dispose-abovebelow-dom-disjoint*:
$nat1\ s1 \Longrightarrow dom\ (dispose1\text{-}above\ f1\ d1\ s1) \cap dom(dispose1\text{-}below\ f1\ d1) = \{\}$
find-theorems - ∩ - = {} *name*:*disjoint name*:*equal*
apply (*subst disjoint-iff-not-equal*)
apply (*rule ballI*)+
unfolding *dispose1-above-def dispose1-below-def*
apply (*simp only*: *l-dom-r-iff*)
using [[*simp-trace*]] **apply** *simp*

done

lemma *f-d1-not-dispose-above* :
$nat1\ s1 \Longrightarrow d1 \notin dom\ (dispose1\text{-}above\ f1\ d1\ s1)$
unfolding *dispose1-above-def*
find-theorems *dom*(- ⩤ -)
by (*simp add*: *l-dom-r-iff*)

lemma *f-d1-not-dispose-below*:
$nat1\text{-}map\ f1 \Longrightarrow nat1\ s1 \Longrightarrow d1 \notin dom\ (dispose1\text{-}below\ f1\ d1)$
unfolding *dispose1-below-def*
find-theorems *dom*(- ⩤ -)

apply (*simp add*: *l-dom-r-iff*)
apply (*rule impI*)

by (*metis f-nat1-map-nat1-elem*)

lemma *f-d1-not-dispose-abovebelow-ext*:
nat1-map f1 $\Longrightarrow$ *sep f1* $\Longrightarrow$ *nat1 s1* $\Longrightarrow$ *d1* $\notin$ *dom* (*dispose1-above f1 d1 s1* $\cup m$ *dispose1-below f1 d1*)
by (*metis UnE f-d1-not-dispose-above f-d1-not-dispose-below k-dispose-abovebelow-dom-disjoint l-dagger-dom munion-def*)

lemma *k-dispose-abovebelow-munion-dom*:
nat1 s1 $\Longrightarrow$ *dom*(*dispose1-above f1 d1 s1* $\cup m$ *dispose1-below f1 d1*)
=
{ $x \in$ *dom f1* . x + *the*(*f1 x*) = *d1* $\vee$ x = *d1* + *s1* }
apply (*rule equalityI*)
apply (*simp-all add*: *l-munion-dom k-dispose-abovebelow-dom-disjoint*)
unfolding *dispose1-above-def dispose1-below-def*
apply (*simp-all add*: *l-dom-r-iff*)
apply (*rule conjI*)
apply (*rule-tac* [*1-*] *subsetI*)
by *auto*

lemma *k-finite-dispose-above*:
finite(*dom f1*) $\Longrightarrow$ *finite* (*dom* (*dispose1-above f1 d1 s1*))
unfolding *dispose1-above-def*
by (*metis finite-Int l-dom-r-iff*)

lemma *k-finite-dispose-below*:
finite(*dom f1*) $\Longrightarrow$ *finite* (*dom* (*dispose1-below f1 d1*))
unfolding *dispose1-below-def*
by (*smt finite-Int l-dom-r-iff*)

lemma *k-finite-dispose-abovebelow-munion*:
finite (*dom f1*) $\Longrightarrow$ *nat1 s1* $\Longrightarrow$ *finite* (*dom* (*dispose1-above f1 d1 s1* $\cup m$ *dispose1-below f1 d1*))
thm *k-finite-munion*[*of dispose1-above f1 d1 s1 dispose1-below f1 d1*]
by (*metis k-dispose-abovebelow-dom-disjoint k-finite-dispose-above k-finite-dispose-below k-finite-munion*)

lemma *k-empty-dispose-above*:
d1 + *s1* $\notin$ *dom f1* $\Longrightarrow$ (*dispose1-above f1 d1 s1*) = *empty*
unfolding *dispose1-above-def*
by (*smt disjoint-iff-not-equal l-dom-r-iff l-map-non-empty-dom-conv mem-Collect-eq*)

lemma *k-nonempty-dispose-below*:
$x \in$ *dom f1* $\Longrightarrow$ x + *the*(*f1 x*) = *d1* $\Longrightarrow$ (*dispose1-below f1 d1*) $\neq$ *empty*
unfolding *dispose1-below-def*
by (*smt dom-def f-in-dom-r-apply-elem mem-Collect-eq*)

lemma *k-dispose1-abovebelow-nonempty*:
nat1 s1 $\Longrightarrow$ *d1* + *s1* $\in$ *dom f1* $\vee$ $x \in$ *dom f1* $\wedge$ x + *the*(*f1 x*) = *d1* $\Longrightarrow$
dispose1-above f1 d1 s1 $\cup m$ *dispose1-below f1 d1* $\neq$ *Map.empty*
apply (*erule disjE*)

apply (*rule notI*)
apply (*simp only*: *dom-eq-empty-conv*[*symmetric*] *k-dispose-abovebelow-munion-dom*)
apply *blast*
by (*metis domIff k-dispose-abovebelow-dom-disjoint k-nonempty-dispose-below l-munion-apply*)

lemma *k-dispose1-abovebelow-empty*:
nat1 s1 $\Longrightarrow$ *sep0* [*d1* $\mapsto$ *s1*] *f1* $\Longrightarrow$ *sep0 f1* [*d1* $\mapsto$ *s1*] $\Longrightarrow$
dispose1-above f1 d1 s1 $\cup m$ *dispose1-below f1 d1* = *Map.empty*
unfolding *sep0-def*
apply (*simp only*: *dom-eq-empty-conv*[*symmetric*] *k-dispose-abovebelow-munion-dom*)
apply *simp*
by *blast*

lemma *k-dispose1-sep0-above-empty*:
sep0 [*d1* $\mapsto$ *s1*] *f1* $\Longrightarrow$ *dispose1-above f1 d1 s1* = *empty*
apply (*simp only*: *dom-eq-empty-conv*[*symmetric*])
unfolding *sep0-def dispose1-above-def*
find-theorems *dom*(- $\lhd$ -)
apply (*simp add*: *dom-eq-empty-conv*[*symmetric*] *l-dom-r-iff*)
by *blast*

lemma *k-dispose1-sep0-below-empty*:
sep0 f1 [*d1* $\mapsto$ *s1*] $\Longrightarrow$ *dispose1-below f1 d1* = *empty*
apply (*simp only*: *dom-eq-empty-conv*[*symmetric*])
unfolding *sep0-def dispose1-below-def*
apply (*simp add*: *dom-eq-empty-conv*[*symmetric*] *l-dom-r-iff*)
by *blast*

lemma *l-dispose1-sep0-above-empty-iff*:
(*dispose1-above f1 d1 s1* = *empty*) = *sep0* [*d1* $\mapsto$ *s1*] *f1*
apply (*rule iffI*)
defer
apply (*rule k-dispose1-sep0-above-empty*,*assumption*)
unfolding *sep0-def dispose1-above-def*
apply (*rule ballI*)
apply *simp*
apply (*rule notI*)
apply (*simp add*: *fun-eq-iff*)
apply (*erule-tac x*=*d1*+*s1* **in** *allE*)
find-theorems (- $\lhd$ -) -
apply (*simp add*: *f-in-dom-r-apply-elem*)
by (*metis domIff*)

lemma *l-dispose1-sep0-below-empty-iff*:
(*dispose1-below f1 d1* = *empty*) = *sep0 f1* [*d1* $\mapsto$ *s1*]
apply (*rule iffI*)
defer
apply (*rule k-dispose1-sep0-below-empty*,*assumption*)
unfolding *sep0-def dispose1-below-def*
apply (*rule ballI*)
apply *simp*
apply (*rule notI*)
apply (*simp add*: *fun-eq-iff*)
apply (*erule-tac x*=*l* **in** *allE*)
find-theorems (- $\lhd$ -) -

apply (*simp add*: *f-in-dom-r-apply-elem*)
by (*metis domIff*)

lemma *f-dispose1-pre-not-in-dom*:
  nat1-map f $\Longrightarrow$ *nat1 s* $\Longrightarrow$ *locs-of d s* $\cap$ *locs f* $= \{\} \Longrightarrow d \notin$ *dom f*
apply (*rule notI*)
find-theorems *name*:*disjoint name*:*iff*
find-theorems - $\in$ *locs-of* - -
find-theorems - $\in$ *locs* -
apply (*simp add*: *disjoint-iff-not-equal*)
apply (*frule f-dom-locs-of*,*assumption*)
apply (*frule f-in-dom-locs*,*assumption*)
apply (*erule-tac x*=*d* **in** *ballE*)
apply (*erule-tac x*=*d* **in** *ballE*)
unfolding *locs-of-def*
by *simp-all*

lemma *l-dispose1-above-singleton*:
  d1+*s1* $\in$ *dom f1* $\Longrightarrow$ *dispose1-above f1 d1 s1* $= [$*d1*+*s1* $\mapsto$ *the*(*f1* (*d1*+*s1*))$]$
unfolding *dispose1-above-def*
apply (*subst fun-eq-iff*)
apply (*rule allI*)
find-theorems (- $\triangleleft$ -) -
unfolding *dom-restr-def*
by *auto*

lemma *l-dispose1-nonempty-above-singleton*:
  dispose1-above f1 d1 s1 $\neq$ *empty* $\Longrightarrow$ *dispose1-above f1 d1 s1* $= [$*d1*+*s1* $\mapsto$ *the*(*f1* (*d1*+*s1*))$]$
by (*metis k-empty-dispose-above l-dispose1-above-singleton*)

lemma *a* $=$ *x* $\Longrightarrow$ $([x \mapsto y]\ a) =$ *Some y*
by *simp*

lemma *a* $\neq$ *x* $\Longrightarrow$ $([x \mapsto y]\ a) =$ *None*
by *simp*

definition
  fbelow :: *F1*
where
  fbelow $\equiv [0 \mapsto 4,\ 5 \mapsto 6,\ 15 \mapsto 3]$

lemma *F1-inv fbelow*
unfolding *fbelow-def F1-inv-defs*
by *auto*

lemma *dispose1-below fbelow 11* $= [5 \mapsto 6]$
unfolding *fbelow-def dispose1-below-def*
apply (*simp add*: *fun-eq-iff*)
apply (*intro conjI allI impI*)
apply (*simp add*: *f-in-dom-r-apply-elem*)
unfolding *dom-restr-def restrict-map-def*
using [[*simp-trace*]] **apply** *simp*

apply *auto*
done

lemma $l \in dom\ fbelow \Longrightarrow l{+}the(fbelow\ l){=}11 \Longrightarrow dispose1\text{-}below\ fbelow\ 11 = [l{\mapsto}the(fbelow\ l)]$
unfolding *fbelow-def dispose1-below-def*
apply *safe*
apply (*simp add*: *fun-eq-iff*)
apply (*intro conjI allI impI*)
apply (*simp-all split*: *split-if-asm*)
unfolding *dom-restr-def restrict-map-def*
apply *simp*
apply *auto*
done

lemma *l-dispose1-below-singleton-useless*:
$l \in dom\ f \Longrightarrow l{+}the(f\ l) = d \Longrightarrow nat1\text{-}map\ f \Longrightarrow sep\ f \Longrightarrow Disjoint\ f \Longrightarrow dispose1\text{-}below\ f\ d = [l \mapsto the(f\ l)]$
unfolding *dispose1-below-def*
find-theorems *simp*:- = (-::('*a* ⇒ '*b*)) -*name*:*HEAP* -*name*:*VDM*
apply (*subst fun-eq-iff*)
apply *simp*
apply (*intro allI impI conjI*)
apply (*simp add*: *f-in-dom-r-apply-the-elem*)
unfolding *dom-restr-def restrict-map-def*
apply (*simp-all*)
apply (*rule impI*)
apply (*erule conjE*)
unfolding *Disjoint-def disjoint-def*
apply (*erule-tac x*=*l* **in** *ballE*)
apply (*erule-tac x*=*x* **in** *ballE*)
find-theorems *locs-of - - ∩ locs-of - -*
apply (*simp-all add*: *l-locs-of-Locs-of-iff*
k-locs-of-arithIff f-nat1-map-nat1-elem)
by (*metis antisym le-iff-add sep-def*)

lemma *l-dispose1-below-singleton-useful*:
$nat1\text{-}map\ f \Longrightarrow sep\ f \Longrightarrow Disjoint\ f \Longrightarrow \forall \cdot\ l \in dom\ f\ .\ l{+}the(f\ l) = d \longrightarrow dispose1\text{-}below\ f\ d = [l \mapsto the(f\ l)]$
by (*metis l-dispose1-below-singleton-useless*)

lemma *l-sum-size-upd*:
$finite(dom\ f) \Longrightarrow x \notin dom\ f \Longrightarrow sum\text{-}size(f(x{\mapsto}y)) = (if\ f = empty\ then\ y\ else\ sum\text{-}size\ f + y)$
unfolding *sum-size-def*
apply *simp*
apply (*intro impI*)
by (*rule setsum-cong*,*simp-all*,*rule impI*,*simp*)
thm *setsum-cong*[*of dom f dom f* (λ *xa* . *the* (*if xa* = *x then Some y else f xa*)) (λ *x* . *the* (*f x*))]

lemma *l-nat1-sum-size-dispose1-ext*:
$nat1\text{-}map\ f1 \Longrightarrow finite\ (dom\ f1) \Longrightarrow sep\ f1 \Longrightarrow nat1\ s1 \Longrightarrow nat1\ (sum\text{-}size\ (dispose1\text{-}ext\ f1\ d1\ s1))$
unfolding *dispose1-ext-def*

apply (*subst l-munion-upd*)
apply (*simp add*: *l-munion-dom k-dispose-abovebelow-dom-disjoint*)
apply (*rule conjI*)
apply (*rule f-d1-not-dispose-above*,*simp*)
apply (*rule f-d1-not-dispose-below*,*simp-all*)
apply (*frule f-d1-not-dispose-abovebelow-ext*[*of f1 s1 d1*],*simp-all*)
apply (*frule k-finite-dispose-abovebelow-munion*[*of f1 s1 d1*],*simp*)
by (*simp add*: *l-sum-size-upd*)

lemma *l-d1-s1-not-dispose1-below*:
  nat1-map f $\Longrightarrow$ *sep f* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *nat1 s* $\Longrightarrow$ $d + s \notin$ *dom* (*dispose1-below f d*)
apply (*cases dispose1-below f d* = *empty*)
apply *simp*
apply (*simp add*: *l-dispose1-sep0-below-empty-iff*[*of f d s*])
unfolding *sep0-def*
apply (*simp*,*erule bexE*)
thm *l-dispose1-below-singleton-useful*
by (*simp add*: *l-dispose1-below-singleton-useful*)

lemma *l-min-loc-dispose1-ext-absorb-above*:
  finite(*dom f*) $\Longrightarrow$ *nat1-map f* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *sep f* $\Longrightarrow$ *nat1 s* $\Longrightarrow$
    min-loc (*dispose1-ext f d s*) = *min-loc*(*dispose1-below f d* $\cup m$ [$d \mapsto s$])
unfolding *dispose1-ext-def*
apply (*cases dispose1-above f d s* = *empty*)
apply (*simp add*: *l-munion-empty-lhs*)
apply (*simp add*: *l-dispose1-nonempty-above-singleton*)
thm *l-munion-commute*[*of* [$d + s \mapsto$ *the* (*f* ($d + s$))] *dispose1-below f d* $\cup m$ [$d \mapsto s$]]
apply (*subst l-munion-commute*)
 apply (*metis* (*full-types*) *k-dispose-abovebelow-dom-disjoint l-dispose1-nonempty-above-singleton nat1-def*)

apply (*subst l-munion-assoc*)
 apply (*metis* (*full-types*) *inf.commute k-dispose-abovebelow-dom-disjoint l-dispose1-nonempty-above-singleton nat1-def*)

  apply (*simp add*: *disjoint-iff-not-equal*)

apply (*subst l-munion-commute*)
back
  apply (*simp add*: *disjoint-iff-not-equal*)
  apply (*subst l-munion-assoc*[*symmetric*])
  apply (*frule f-d1-not-dispose-below*,*simp-all*)

find-theorems *min-loc* (- $\cup m$ -)
thm *k-min-loc-munion-singleton*[*of dispose1-below f d* $\cup m$ [$d \mapsto s$] $d + s$ *the* (*f* ($d + s$))]
apply (*subst k-min-loc-munion-singleton*)
  apply (*rule k-finite-munion*, *simp-all*)
  apply (*metis k-finite-dispose-below*)
  apply (*metis f-d1-not-dispose-below nat1-def*)
  apply (*subst l-munion-dom*)
    apply (*frule f-d1-not-dispose-below*,*simp-all add*: *l-d1-s1-not-dispose1-below*)
apply (*intro conjI impI*)
apply (*simp add*: *l-munion-singleton-not-empty f-d1-not-dispose-below*)
apply (*cases dispose1-below f d* = *empty*)

apply (*simp add*: *l-munion-empty-lhs l-min-loc-singleton*)
apply (*simp add*: *l-dispose1-sep0-below-empty-iff*[*of f d s*])
unfolding *sep0-def*
apply *simp*
apply (*erule bexE*)
apply (*simp add*: *l-dispose1-below-singleton-useless*) — so the useless version works?! hum...
apply (*subst k-min-loc-munion-singleton*)
  apply (*metis finite-singleton*)
  apply (*frule f-nat1-map-nat1-elem*,*simp-all*)
  apply (*metis sep-def*)
apply (*subst l-munion-dom*)
  apply (*frule f-nat1-map-nat1-elem*,*simp-all*)
  apply (*metis sep-def*)
done

lemma *l-sep0-dispose1-abovebelow-ext*:
 finite(*dom f1*) ⟹ *nat1-map f1* ⟹ *Disjoint f1* ⟹ *sep f1* ⟹ *nat1 s1* ⟹
  sep0 ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1*)
    [*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1 s1*)]
unfolding *sep0-def*
apply (*rule ballI*)
apply (*simp add*: *l-min-loc-dispose1-ext-absorb-above*)
find-theorems - ∈ *dom*(- -◁ -)
apply (*simp add*: *f-in-dom-ar-subsume f-in-dom-ar-the-subsume*)
apply (*cases dispose1-below f1 d1 = empty*)
apply (*simp add*: *l-min-loc-singleton l-munion-empty-lhs*)
apply (*metis k-nonempty-dispose-below l-dom-ar-not-in-dom*)

  apply (*simp add*: *l-dispose1-sep0-below-empty-iff*[*of f1 d1 s1*])
  unfolding *sep0-def*
  apply *simp*
  apply (*erule bexE*)
  apply (*simp add*: *l-dispose1-below-singleton-useless*) — so the useless version works?! hum...
  apply (*subst k-min-loc-munion-singleton*,*simp-all*)
    apply (*metis sep-def*)
    apply (*metis l-dom-ar-notin-dom-or le-add1 min-max.le-iff-inf sep-def*)
done

lemma *l-sep0-dispose1-ext-abovebelow*:
  finite(*dom f1*) ⟹ *nat1-map f1* ⟹ *Disjoint f1* ⟹ *sep f1* ⟹ *nat1 s1* ⟹
   sep0 [*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1 s1*)]
    ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1*)
unfolding *sep0-def*
apply (*rule ballI*)
apply (*simp add*: *l-min-loc-dispose1-ext-absorb-above*)
find-theorems *simp*:- ∈ *dom*(- -◁ -)
apply (*subst l-dom-ar-not-in-dom*,*simp-all*)
unfolding *dispose1-ext-def*
apply (*subst l-sum-size-munion-singleton*)
  apply (*metis k-finite-dispose-abovebelow-munion nat1-def*)
  apply (*simp add*: *k-dispose-abovebelow-dom-disjoint f-d1-not-dispose-abovebelow-ext*)
apply (*cases dispose1-below f1 d1 = empty*)
apply (*simp add*: *l-min-loc-singleton l-munion-empty-rhs l-munion-empty-lhs*)
  apply (*intro conjI impI*)
  apply (*simp add*: *l-dispose1-sep0-above-empty-iff*)
 apply (*metis k-dispose1-abovebelow-empty k-dispose1-abovebelow-nonempty l-dispose1-sep0-below-empty-iff*

```
nat1-def)
  apply (simp add: l-dispose1-nonempty-above-singleton l-sum-size-singleton)
  apply (simp add: l-dispose1-sep0-above-empty-iff)
  unfolding sep0-def sep-def
  apply simp
  apply (rule notI)
  apply (erule-tac x=d1+s1 in ballE,simp-all)
  apply smt
  apply (fold sep-def)
apply (cases dispose1-above f1 d1 s1 = empty)
apply (simp add: l-min-loc-singleton l-munion-empty-rhs l-munion-empty-lhs)
apply (simp add: l-dispose1-sep0-above-empty-iff
            l-dispose1-sep0-below-empty-iff[of f1 d1 s1])
  unfolding sep0-def
  apply simp
  apply (erule bexE)
    thm l-dispose1-below-singleton-useful
        k-min-loc-munion-singleton[simplified]
  apply (simp add: l-dispose1-below-singleton-useful
                   l-sum-size-singleton)
  apply (subst k-min-loc-munion-singleton)
    apply (metis finite-singleton)
    apply (frule f-nat1-map-nat1-elem,simp)
      apply (simp add: disjoint-iff-not-equal)
    apply (frule f-nat1-map-nat1-elem,simp)
    unfolding min-def
    apply (simp)
    unfolding sep-def
    apply (rule notI)
    apply (erule-tac x=la in ballE,simp-all)
    apply smt   — ???? How is this being proved ????
    apply (fold sep-def)
apply (simp add: l-munion-empty-iff k-dispose-abovebelow-dom-disjoint)
  apply (simp add: l-dispose1-nonempty-above-singleton l-sum-size-singleton)
  apply (simp add: l-dispose1-sep0-above-empty-iff
                   l-dispose1-sep0-below-empty-iff[of f1 d1 s1])
  unfolding sep0-def
  apply simp
  apply (erule bexE)
  thm k-min-loc-munion-singleton[simplified]
  apply (frule f-nat1-map-nat1-elem,simp)
    back
  apply (simp add: l-dispose1-below-singleton-useful
                   l-sum-size-munion-singleton l-sum-size-singleton
                   k-min-loc-munion-singleton)
  unfolding min-def sep-def
  apply simp
  apply (rule notI)
  apply (erule-tac x=d1+s1 in ballE,simp-all)
  apply smt   — ???? How is this being proved ????
done
```

```
lemma l-disjoint-dispose1-ext:
   finite(dom f1) ⟹ nat1-map f1 ⟹ Disjoint f1 ⟹ sep f1 ⟹ nat1 s1 ⟹ dispose1-pre f1 d1 s1
```

```
⟹
    disjoint (locs-of (min-loc (dispose1-ext f1 d1 s1)) (HEAP1.sum-size (dispose1-ext f1 d1 s1)))
      (locs ((dom (dispose1-below f1 d1) ∪ dom (dispose1-above f1 d1 s1)) -◁ f1))
apply (simp add: l-min-loc-dispose1-ext-absorb-above)
find-theorems simp:(- -◁ -)
thm l-locs-dom-ar-iff l-dom-ar-accum
apply (simp add: l-dom-ar-accum[symmetric])
unfolding disjoint-def dispose1-ext-def dispose1-pre-def
apply (cases dispose1-below f1 d1 = empty)
apply (simp add: l-munion-empty-lhs l-min-loc-singleton l-munion-empty-rhs l-dom-ar-none)
  apply (cases dispose1-above f1 d1 s1 = empty)
    apply (simp add: l-munion-empty-lhs l-sum-size-singleton l-dom-ar-none)
  apply (simp add: l-dispose1-nonempty-above-singleton l-dispose1-sep0-above-empty-iff
              l-sum-size-munion l-sum-size-singleton)
  unfolding sep0-def
  apply (simp add: l-locs-dom-ar-iff)
  apply (simp add: disjoint-iff-not-equal)
  apply (rule ballI)+
  apply (frule f-nat1-map-nat1-elem,simp)
  unfolding locs-of-def
  apply simp
  apply (fold locs-of-def)
  apply smt

  apply (cases dispose1-above f1 d1 s1 = empty)
    apply (simp add: l-munion-empty-lhs l-sum-size-singleton l-dom-ar-none)
    apply (simp add: l-dispose1-sep0-below-empty-iff[of f1 d1 s1])
    unfolding sep0-def
    apply simp
    apply (erule bexE)
    apply (simp add: l-dispose1-below-singleton-useful)
      thm l-sum-size-munion-singleton[simplified] l-sum-size-singleton
            k-min-loc-munion-singleton[simplified]
            l-dispose1-nonempty-above-singleton
    apply (subst k-min-loc-munion-singleton)
      apply (metis finite-singleton)
      apply (simp add: disjoint-iff-not-equal)
      apply (metis sep-def)
    apply (subst l-sum-size-munion-singleton)
      apply (metis finite-singleton)
      apply (simp add: disjoint-iff-not-equal)
      apply (metis sep-def)
    apply (simp add: l-sum-size-singleton l-locs-dom-ar-iff)

    apply (simp add: disjoint-iff-not-equal)
    apply (rule ballI)+
    apply (frule f-nat1-map-nat1-elem,simp)
    unfolding locs-of-def
    apply simp
    apply (fold locs-of-def)
    apply smt

  apply (simp add: l-dispose1-nonempty-above-singleton l-dispose1-sep0-above-empty-iff
              l-dispose1-sep0-below-empty-iff[of f1 d1 s1]
              l-sum-size-munion l-sum-size-singleton)
  unfolding sep0-def
```

apply *simp*
apply (*erule bexE*)
apply (*simp add*: *l-dispose1-below-singleton-useful*)
apply (*subst k-min-loc-munion-singleton*)
apply (*metis finite-singleton*)
apply (*simp add*: *disjoint-iff-not-equal*)
apply (*metis sep-def*)
apply (*simp add*: *min-def*)
apply (*subst l-sum-size-munion-singleton*)
apply (*metis* (*lifting*) *k-finite-dispose-abovebelow-munion l-dispose1-above-singleton l-dispose1-below-singleton-useless nat1-def*)

apply (*simp add*: *disjoint-iff-not-equal*)
apply (*rule ballI*)+
apply (*simp add*: *l-munion-dom*)
apply (*metis sep-def*)

apply (*simp add*: *l-munion-empty-iff*)
apply (*subst l-sum-size-munion-singleton*)
apply (*metis finite-singleton*)
apply (*simp add*: *disjoint-iff-not-equal*)
apply (*simp add*: *l-sum-size-singleton l-dom-ar-accum l-locs-dom-ar-general-iff*)

— Perhaps use l_right_diff_left_dist? Nah... just follow previous strategy
apply (*simp add*: *disjoint-iff-not-equal*)
apply (*rule ballI*)+
apply (*frule f-nat1-map-nat1-elem*,*simp*)
apply (*frule f-nat1-map-nat1-elem*)
apply *simp* **back**
unfolding *locs-of-def*
apply *simp*
apply *smt*
done

find-theorems - ∈ *locs* -

lemma *l-locs-maximal-quickspec*:
(*locs f*) -◁ *f* = (*locs g*) -◁ *g*
oops

lemma *l-locs-maximal-quickspec*:
(*locs f*) -◁ *f* = *empty*
oops

lemma *l-locs-empty-quickspec*:
(*locs empty* = {})
oops

find-theorems *locs empty*

end

theory *HEAP1Proofs*

imports *HEAP1 HEAP1Lemmas*
begin

Add lemmas k_in_dom_locs = l_in_dom_locs for when the same lemma ("l_") has multiple uses in a theory?

E.7 NEW 1 proofs

As part of the strategy for mechanisation with sledgehammer we rely on a few patterns for "zooming", "witnessing", "bridging", and "weakening". To easily identify what lemma participate in what pattern, we use some name conventions as below. Prefixes can be combined to indicate patterns are being combined.

1. Zooming: lemma mames prefixed with "z_"

   Pattern that takes into account a (sub-)set of definitions of interest to unfold and tackle at different stages. These are problem dependant and require expert annotation (of defs?).

   The pattern is applied by decomposing the goal, top-down, into its suggoal parts declared as lemmas with appropriate instantiations.

   It achieves separation of concerns given one concentrate at the right level of abstraction during a proof.

2. Witnessing: lemma mames prefixed with "w_"

   Type1: strip defs the user tagged to; try and get 1-point-rule to work Type2: Type1 where you need an explicit instantiation from user

   Most POs involve instantiating some (difficult) existential quantifier or interest. With this pattern we instantiate variables to uninterpreted constants following by the application of the zooming pattern. On many models, this leaves to obvious witness to choose under certain conditions, to be added as lemmas for the subcases of interest given the model at hand.

   Another approach is to instantiate the quantified after state as simply the before state (i.e. as if we were dealing with a SKIP-OP). This is clearly wrong, yet after (safe-)simplification often gives insight into what the correct (or approximate) instantiation should be. This is useful to when the model does not provide equations for the quantified after state.

   For instance, we use uninterpreted witnessing for the proof of NEW1 feasibility. This leads to the instantiations of the suggested lemmas *zw_new1_post* and *zw_F1_inv*.

3. Bridges: lemma mames prefixed with "b_"

   Certain information about types and predicates (e.g. invariant, pre/post) are "obvious" yet not immediately known/available to Isabelle. The choice to what is to be put into the "goal context" by default requires some practice, yet is pretty deterministic: all the type-related parts of goals that keep occurring in the middle of proofs, yet are not the relevant goal to be proved.

   For these scenarios, we add type or definition "bridges" that tell Isabelle to take them (or a variation of them) into account during simplification (i.e. declare some tags to definitions like *intro*).

   For instance, lemmas are needed to prove the feasibility of NEW1. They all require some knowledge about the before state invariant and the precondition under the appropriate instantiations, or the fact the map f is finite and with a nat1 range. We add these as lemmas below to ensure these required information is not hampering automation.

4. Weakening: lemma mames prefixed with "k_"

   One usually do not have enough information about goals function symbols in order to directly discharge them. Adding specific lemmas to that effect is often unlikely (and leads to lemmas that are too specific to be reused).

   Instead, we often need lemmas that much at specific parts of the goal (backward chaining) or at specific part of the hypothesis (forward chaining) to weaken the overall task to pieces manageable by the theorem prover.

TODO: explain Naur's N-Queen approach to explaining the problem!

E.7.1 NEW 1 FSB

These lemmas rely on general (expert) lemmas about maps and Other mathematical toolkit operatos, many of which Isabelle already has useful lemmas for.

In this development, we need to create these from scratch. Yet, although a bit artificial, we shield the development from these general goals/proofs by having them in a separate theory.

In practice, we antecipate that these lemmas will be reused in other VDM-style map problems. As indeed is already evident from the various lemmas "stolen" from ZEves' mathematical toolkit (i.e. the FM style of model and proof transfer across provers too). Or else, we might be having some outcome bias, given authors expertise in this other prover. Either way, it does show that proof patterns do exist beyond specific provers and examples.

E.7.1.1 NEW 1 FSB weakening lemmas for equal case

For *new*1_*eq* case lemmas are easier: we just need to show the submap satisfy the various parts of the state invariant. We prove a lemma for each such subpart below. They follow directly from general lemmas about the involved operators and are all sledgehammered.

To allow for our lemma collection/analysis tool to work, we avoid (in X) and explicitly collect the locale-specific lemmas.

context *level1-new*
begin

lemma *k-new1-Disjoint-dom-ar*:
Disjoint ({*x*} -⊲ *f1*)
by (*metis F1-inv-def k-Disjoint-dom-ar l1-invariant-def*)

lemma *k-new1-sep-dom-ar*:
sep ({*r*} -⊲ *f1*)
by (*metis F1-inv-def k-sep-dom-ar l1-invariant-def*)

lemma *k-new1-nat1-map-dom-ar*:
nat1-map ({*r*} -⊲ *f1*)
by (*metis F1-inv-def k-nat1-map-dom-ar l1-invariant-def*)

lemma *k-new1-finite-dom-ar*:
finite (*dom* ({*r*} -⊲ *f1*))
by (*metis F1-inv-def k-finite-dom-ar l1-invariant-def*)

E.7.1.2 NEW 1 FSB weakening lemmas for greater than case

For *new*1_*gr* case lemmas are not as easy. Our definition of VDM map union rely on a side condition about disjointness of map's domains, which will feature in all proofs for *new*1_*gr*.

Historically, we had made a mistake (oops): we defined the models in Isabelle using a version of VDM dagger (- † - or - ++ - in Isabelle) instead of map union. After correcting the mistake we had a throve of lemmas for dagger, which are useful for proving map union, so we kept both.

Isabelle does not have map union but (Isabelle) map update (- ++ -). We define VDM map union with map update where domains are disjoint, or undefined otherwise. Thus, having had these lemmas about map update were quite useful for a general strategy for proving VDM map union in Isabelle (with this encoding): prove it for dagger then establish the disjointness of domains for the maps involved and it does work, in most cases (i.e. an example where it does not occurs for certain algebraic rules about our *locs* function, see below in ???).

Given that, as before for *new_eq*, we show show the submap (-⩤-) updated (-†-) or extended (- ∪*m* -) satisfy the various parts of the state invariant. We prove a lemma for each such subpart below. They follow directl from general lemmas about the involved operators and are all sledgehammered.

lemma *k-new1-Disjoint-dom-ar-munion*:
$r \in dom\ f1 \Longrightarrow the\ (f1\ r) > s1 \Longrightarrow Disjoint\ (\{r\} \lhd\!\!- f1 \cup m\ [r + s1 \mapsto the\ (f1\ r) - s1])$
by (*smt F1-inv-def k-Disjoint-dom-ar-munion l1-input-notempty-def l1-invariant-def*)

lemma *k-new1-sep-dom-ar-munion*:
$r \in dom\ f1 \Longrightarrow the\ (f1\ r) > s1 \Longrightarrow sep\ (\{r\} \lhd\!\!- f1 \cup m\ [r + s1 \mapsto the\ (f1\ r) - s1])$
by (*smt F1-inv-def k-sep-dom-ar-munion l1-input-notempty-def l1-invariant-def*)

lemma *k-new1-nat1-map-dom-ar-munion*:
$r \in dom\ f1 \Longrightarrow the\ (f1\ r) > s1 \Longrightarrow nat1\text{-}map\ (\{r\} \lhd\!\!- f1 \cup m\ [r + s1 \mapsto the\ (f1\ r) - s1])$
by (*metis F1-inv-def k-nat1-map-dom-ar-munion l1-invariant-def l1-input-notempty-def*)

lemma *k-new1-finite-dom-ar-munion*:
$r \in dom\ f1 \Longrightarrow the\ (f1\ r) > s1 \Longrightarrow finite\ (dom(\{r\} \lhd\!\!- f1 \cup m\ [r + s1 \mapsto the\ (f1\ r) - s1]))$
by (*metis* (*mono-tags*) *F1-inv-def k-finite-dom-ar-munion l1-input-notempty-def l1-invariant-def*)

E.7.1.3 NEW 1 FSB goal-splitting lemmas

From the top-down strategy for the feasibility proof, we need to provide zooming-weakening lemmas to enable sledgehammer to work for our given witnesses, which also determine the key step in the proof: the splitting of cases for exact and surplus memory allocation.

As it happened for the invariant parts for each case, these lemmas operate on each part of the feasibility proof this time, namely the postcondition, the state invariant and the outputs. Obviously, the zooming strategy works well given this setup since the lemmas above are already in the shape needed.

That is, when working top-down as we did, the unpicking of the various parts of the feasibility proof obligation leads to the suggestion of these lemma shapes up to the point where available (and general) mathematical toolkit lemmas apply, modulo a few new ones needed. That's usually where expert input is needed.

Call it what you like, this top-down strategy/pattern/tactic, repeats across problems in the formal methods domain, where automation depends on the quality and shape of the general lemmas available. Our hope is that, with enough data about expert choices regarding specialised versions of general lemmas (as well as new general lemmas themselves), AI4FM tools would be able to spot the similarities/features/patterns and suggest them to new/novice users.

lemma *zw-new1-post-eq*:
$r \in dom\ f1 \Longrightarrow the\ (f1\ r) = s1 \Longrightarrow new1\text{-}post\text{-}eq\ f1\ s1\ (\{r\} \lhd\text{-} f1)\ r$
unfolding *new1-post-eq-def*
by *auto*

lemma *zw-F1-inv-new1-eq* :
$r \in dom\ f1 \Longrightarrow the\ (f1\ r) = s1 \Longrightarrow F1\text{-}inv\ (\{r\} \lhd\text{-} f1)$
by (*metis F1-inv-def k-new1-Disjoint-dom-ar k-new1-finite-dom-ar k-new1-nat1-map-dom-ar k-new1-sep-dom-ar*)

lemma *zw-new1-post-gr*:
$r \in dom\ f1 \Longrightarrow the\ (f1\ r) > s1 \Longrightarrow new1\text{-}post\text{-}gr\ f1\ s1\ (\{r\} \lhd\text{-} f1 \cup m\ [r + s1 \mapsto the\ (f1\ r) - s1])\ r$
unfolding *new1-post-gr-def*
by *auto*

lemma *zw-F1-inv-new1-gr* :
$r \in dom\ f1 \Longrightarrow the\ (f1\ r) > s1 \Longrightarrow F1\text{-}inv\ (\{r\} \lhd\text{-} f1 \cup m\ [r + s1 \mapsto the\ (f1\ r) - s1])$
by (*metis* (*full-types*) *F1-inv-def*
k-new1-Disjoint-dom-ar-munion
k-new1-finite-dom-ar-munion
k-new1-nat1-map-dom-ar-munion
k-new1-sep-dom-ar-munion)

E.7.1.4 NEW 1 FSB main theorem

Finally, the top-down strategy applies zomming and weakening patterns, once the key point about splitting exact and surplus memory allocation is observed[1].

theorem *locale1-new-FSB*: *PO-new1-feasibility*
unfolding *PO-new1-feasibility-def new1-postcondition-def*
apply (*insert l1-new1-precondition-def*)
unfolding *new1-pre-def new1-post-def*
apply (*erule bexE*)
find-theorems - $\leq$ - = ((- < -) $\vee$ -)
apply (*simp only*: *le-eq-less-or-eq*)
apply (*erule disjE*)
apply (*metis zw-new1-post-gr zw-F1-inv-new1-gr*)
apply (*metis zw-new1-post-eq zw-F1-inv-new1-eq*)
done

end

E.8 DISPOSE 1 proofs

The strategy for the finiteness proof was the first one to be constructed. It generated various lemmas in different theories, some general missing lemmas about maps, other problem-specific lemmas missing that are useful for other goals.

We had various attempts and they operate on the main function symbols in different order. The bottom line is the case analysis around the DISPOSE1 auxiliary functions being empty or not. After finishing the proof, we minimised the number of lemmas needed as much as possible by cleaning up / deleting *unused_thms*.

The proofs rely entirely on the ability to distribute over munion, which requires the side condition that domains involved are disjoint. This is the hard part on all invariant proofs, which has been extracted as a lemma, namely *l_dispose*1*_munion_disjoint*.

context *level1-dispose*

[1] Oddly enough, we are saying "finally" for where usually is the place work begins!

begin

lemma *k-dispose-above-ext-dom-disjoint-aux*:
d1 ∉ *dom* (*dispose1-above f1 d1 s1*)
by (*metis f-d1-not-dispose-above l1-input-notempty-def*)

lemma *k-dispose-below-ext-dom-disjoint-aux*:
d1 ∉ *dom* (*dispose1-below f1 d1*)
by (*metis f-d1-not-dispose-below l1-invariant-def F1-inv-def l1-input-notempty-def*)

lemma *k-finite-dispose-above-aux*:
finite (*dom* (*dispose1-above f1 d1 s1*))
by (*metis f-F1-inv-finite k-finite-dispose-above l1-invariant-def*)

lemma *k-finite-dispose-below-aux*:
finite (*dom* (*dispose1-below f1 d1*))
by (*metis f-F1-inv-finite k-finite-dispose-below l1-invariant-def*)

Now for the *KEY* lemma, which is used on all F1_inv DISPOSE1 proofs! It was discovered during the finiteness proof (the first part of the invariant tackled). It was then used for nat1_map and sep (and possibly Disjoint).

Still, through the proof for sep, we found that there is an underlying lemma within this one, which is about the possible values for min_loc (l_min_loc_dispose1_ext_iff). These values underlie the complicated case analysis here.

TODO: we could refactor this proof in terms of the one for min_loc, yet we will keep it as is as an example of how these more complex lemmas come to the surface.

Therefore, this lemma is the weakening rule to enable the application of various operators over map union, whereas the one on min_loc performs the appropriate case analysis.

lemma *l-dispose1-munion-disjoint*:
dom ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -⊲ *f1*) ∩
dom [*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1 s1*)] = {}
— simp would do as well
find-theorems - ∩ - = {}
apply (*simp add*: *l-dom-dom-ar*)
— simp alone already simplified goal; LEMMA about dom_ar improves on result
unfolding *dispose1-ext-def*
apply (*rule impI*)
apply (*cases dispose1-below f1 d1 = empty*) — prefer cases instead of subgoal_tac
apply (*simp-all add*: *l-munion-empty-rhs*)
apply (*cases dispose1-above f1 d1 s1= empty*)
apply (*simp-all add*: *l-munion-empty-lhs*)
— nothing to adjoin: below=above=empty
apply (*simp add*: *l-min-loc-singleton*)
apply (*insert l1-dispose1-precondition-def*)
unfolding *dispose1-pre-def disjoint-def*
apply (*insert l1-input-notempty-def*)
apply (*insert l1-invariant-def*)
apply (*frule f-F1-inv-nat1-map*)
apply (*simp add*: *f-dispose1-pre-not-in-dom*)
— above to adjoin: below=empty; not above = empty
find-theorems *min-loc* (- ∪*m* -)
thm *k-min-loc-munion-singleton*[*of dispose1-above f1 d1 s1 d1 s1*]
find-theorems *name*:*contrapos*

thm *k-min-loc-munion-singleton*[*THEN subst*, *of dispose1-above f1 d1 s1 d1 s1* (λ *x* . *x* ∈ *dom f1*)]
apply (*simp add*: *k-min-loc-munion-singleton*[*simplified*]

```
                k-finite-dispose-above-aux
                k-dispose-above-ext-dom-disjoint-aux
          split: split-if-asm)
  apply (simp add: l-dispose1-sep0-above-empty-iff)
  unfolding sep0-def
  apply (simp add: l-dispose1-above-singleton)
  unfolding min-def
   thm f-dispose1-pre-not-in-dom[of f1 s1 d1]
  apply (simp split: split-if-asm
          add: f-dispose1-pre-not-in-dom
                f-F1-inv-nat1-map
                 l1-invariant-def)
 — below to adjoin: not below = empty; above=empty
 apply (cases dispose1-above f1 d1 s1= empty)
 apply (simp-all add: l-munion-empty-lhs)
  thm k-min-loc-munion-singleton[THEN subst, of dispose1-below f1 d1 d1 s1 (λ x . x ∈ dom f1)]
  apply (simp add: k-min-loc-munion-singleton[simplified]
                k-finite-dispose-below-aux
                k-dispose-below-ext-dom-disjoint-aux
          split: split-if-asm)
  apply (simp add: l-dispose1-sep0-above-empty-iff)
  unfolding sep0-def
  apply (simp add: l-dispose1-above-singleton)
  unfolding min-def
   thm f-dispose1-pre-not-in-dom[of f1 s1 d1]
  apply (simp split: split-if-asm
          add: f-dispose1-pre-not-in-dom
                f-F1-inv-nat1-map
                 l1-invariant-def)
  apply (metis Min-in dom-eq-empty-conv k-finite-dispose-below-aux) — TODO: study Min_in interpret proof!
 — both to adjoin: not below = above = empty
  — NOTE: unfortunately, because dispose1_below has a free variable l, we need something different
  apply (simp add: l-dispose1-sep0-below-empty-iff[of f1 d1 s1])
  apply (frule l-dispose1-nonempty-above-singleton)
  unfolding sep0-def
  apply simp
  unfolding F1-inv-def
  apply (elim conjE bexE)
  thm l-dispose1-below-singleton-useful
  apply (frule l-dispose1-below-singleton-useful)
  apply assumption+
  apply (erule-tac x=l in ballE)
  apply (erule impE)
  apply (simp-all)+ — Funny: simp_all doesn't quite work here
  find-theorems min-loc(- ∪m -)
 thm k-min-loc-munion-singleton[of [d1 + s1 ↦ the (f1 (d1 + s1))] ∪m [l ↦ the (f1 l)],simplified]
  apply (frule f-dispose1-pre-not-in-dom,simp-all)
  apply (frule f-nat1-map-nat1-elem)
  apply simp
   back

  unfolding munion-def
  apply (simp add: l-dagger-dom)
  unfolding min-loc-def
  apply (simp add: l-dagger-dom split: split-if-asm)
```

apply *smt*
by (*metis l-dagger-not-empty map-upd-nonempty*)

lemma *z-F1-inv-dispose1-finite*:
finite (*dom* ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1* ∪*m*
[*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1 s1*)]))
find-theorems *simp*:*finite*(*dom*(- ∪*m* -))
apply (*rule k-finite-munion*)
apply (*metis F1-inv-def finite-Diff l1-invariant-def l-dom-dom-ar*)
apply (*metis dom-eq-singleton-conv finite.simps*)
by (*rule l-dispose1-munion-disjoint*)

lemma *z-F1-inv-dispose1-nat1-map*:
nat1-map
((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1* ∪*m*
[*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1 s1*)])
find-theorems *simp*: - ∪*m* -
find-theorems *nat1-map* (- ∪*m* -)
thm *l-munion-dom-ar-assoc*[*of* (*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) *f1*
[*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1*
s1)]]
find-theorems *simp*: *nat1-map*(- -◁ -)
find-theorems *nat1-map* [- ↦ -]
apply (*rule l-nat1-map-munion*)
apply (*metis f-F1-inv-nat1-map k-nat1-map-dom-ar l1-invariant-def*)
apply (*metis F1-inv-def l1-input-notempty-def l1-invariant-def l-nat1-map-singleton l-nat1-sum-size-dispose1-ext*)
by (*metis l-dispose1-munion-disjoint*)

lemma *z-F1-inv-dispose1-sep*:
sep ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1* ∪*m*
[*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1 s1*)])
find-theorems *sep* (- ∪*m* -)
apply (*rule l-sep-munion*)
apply (*metis l-dispose1-munion-disjoint*)
apply (*metis f-F1-inv-sep k-sep-dom-ar l1-invariant-def*)
apply (*metis F1-inv-def l1-input-notempty-def l1-invariant-def l-nat1-sum-size-dispose1-ext l-sep-singleton*)
apply (*insert l1-invariant-def*)
apply (*insert l1-input-notempty-def*)
apply (*frule-tac* [*1-2*] *f-F1-inv-finite*)
apply (*frule-tac* [*1-2*] *f-F1-inv-nat1-map*)
apply (*frule-tac* [*1-2*] *f-F1-inv-disjoint*)
apply (*frule-tac* [*1-2*] *f-F1-inv-sep*)
by (*simp-all add*: *l-sep0-dispose1-abovebelow-ext l-sep0-dispose1-ext-abovebelow*)

lemma *z-F1-inv-dispose1-Disjoint*:
Disjoint
((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1* ∪*m*
[*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1 s1*)])
find-theorems *simp*:*Disjoint*(- ∪*m* -)

apply (*rule l-disjoint-singleton-upd*)
apply (*metis f-F1-inv-nat1-map k-nat1-map-dom-ar l1-invariant-def*)
apply (*smt Collect-empty-eq Un-empty-left dom-eq-singleton-conv inf-commute*
inf-sup-absorb insert-absorb insert-def l-dispose1-munion-disjoint singleton-conv2 sup-commute)
apply (*metis F1-inv-def l1-input-notempty-def l1-invariant-def l-nat1-sum-size-dispose1-ext*)
apply (*metis f-F1-inv-disjoint k-Disjoint-dom-ar l1-invariant-def*)
by (*metis F1-inv-def l1-dispose1-precondition-def l1-input-notempty-def l1-invariant-def l-disjoint-dispose1-ext*)

lemma *z-F1-inv-dispose1-post* :
F1-inv ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1* ∪*m*
[*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1 s1*)])
by (*metis F1-inv-def*
z-F1-inv-dispose1-Disjoint
z-F1-inv-dispose1-finite
z-F1-inv-dispose1-nat1-map
z-F1-inv-dispose1-sep)

theorem *locale1-dispose-FSB*: *PO-dispose1-feasibility*
unfolding *PO-dispose1-feasibility-def dispose1-postcondition-def* — dispose1_post_def
apply (*simp add*: *dispose1-equiv*)
— Apply equivalence LEMMA to tame the proof
unfolding *dispose1-post2-def*
by (*simp add*: *z-F1-inv-dispose1-post*)

end

end

theory *HEAP1SanityProofs*
imports *HEAP1Sanity HEAP1Proofs*
begin

E.9 Proof of some properties of interest

E.9.1 Invariant testing

lemma *l-F1-inv-example*: *F1-ex-inv F1-ex*
unfolding *F1-ex-inv-defs* **by** *auto*

lemma *F1-inv* [*0* ↦ *4*, *5* ↦ *11*]
unfolding *F1-inv-defs* **by** *auto*

lemma *F1-inv* [*0* ↦ *4*, *10* ↦ *6*, *20* ↦ *2*]

unfolding *F1-inv-defs* **by** *auto*

lemma ¬ *F1-inv* [*0* ↦ *4*, *4* ↦ *11*]
unfolding *F1-inv-defs* **by** *auto*

E.9.2 Operations properties

E.9.2.1 NEW 1 shrinks the memory

context *level1-new*
begin*f1′* ⊆$_m$ *f1 not true of course on the new_gr* **lemma** *new1-postcondition-state-changes-headon*:
PO-new1-postcondition-state-changes r
unfolding *PO-new1-postcondition-state-changes-def new1-postcondition-def new1-post-defs*
apply (*intro allI impI*)
apply (*elim conjE disjE*)
apply (*simp-all* (*no-asm-simp*) *add*: *l-map-dom-ar-neq*)
apply (*insert l1-invariant-def*)
apply (*insert l1-input-notempty-def*)
unfolding *F1-inv-def*
apply (*elim conjE*)
apply (*subst fun-eq-iff*)
apply (*insert l-disjoint-mapupd-keep-sep*[*of f1 r s1*])
apply (*simp* (*no-asm-simp*))
apply *simp*
apply (*subgoal-tac dom* ({*r*} -◁ *f1*) ∩ *dom* [*r* + *s1* ↦ *the* (*f1 r*) - *s1*] = {})
apply (*simp add*: *l-munion-apply*)
unfolding *dom-antirestr-def*
by *auto*

lemma *new1-postcondition-state-locs-subset-headon*:
PO-new1-postcondition-state-locs-subset r
unfolding *PO-new1-postcondition-state-locs-subset-def*
apply (*intro allI impI*)
apply (*insert l1-invariant-def*)
apply (*insert l1-input-notempty-def*)
— prepare goal: add invariants
apply (*rule subsetI*)
— prepare goals: expand main ops
— G1: locs-subset
unfolding *new1-postcondition-def F1-inv-def locs-def*
apply *simp-all*
— prepare goals: expand main defs (no disjunctions)
unfolding *new1-post-defs*
apply (*elim conjE disjE bexE*)
— prepare goal: flatten assumptions of G1
apply (*metis f-in-dom-ar-subsume f-in-dom-ar-the-subsume*)
— G1.1: new1_eq locs-subset
apply (*subgoal-tac dom* ({*r*} -◁ *f1*) ∩ *dom* [*r*+*s1* ↦ *the*(*f1 r*) - *s1*] = {})
— G1.2: new1_gr locs-subset (assuming munion WD)
apply (*simp add*: *l-munion-dom l-munion-apply*)
apply (*insert l-disjoint-mapupd-keep-sep*[*of f1 r s1*])
apply (*erule disjE*)
apply *simp-all*
apply (*rule-tac x*=*r* **in** *bexI*)
apply (*smt b-new1-gr-upd-within-req-size l1-input-notempty-def set-mp*)
— G1.2.1: new1_gr locs-subset RHS-munion
apply *assumption*

```
    — G1.2.1.1: bexI impI of G1.2.1
  apply (split split-if-asm)
   apply simp
    — G1.2.1: repeated because of the if-asm
  apply (metis f-in-dom-ar-subsume f-in-dom-ar-the-subsume)
   — G1.2.2: new1_gr locs-subset LHS-munion
  apply (metis f-in-dom-ar-subsume)
   — G1.2.3: subgoal-tac discharge
done

lemma new1-postcondition-state-locs-subset-planned:
 PO-new1-postcondition-state-locs-subset r
unfolding PO-new1-postcondition-state-locs-subset-def
apply (intro allI impI)
unfolding new1-postcondition-def F1-inv-def locs-def
apply (elim conjE)
apply simp
apply (intro conjI impI)
defer 1
apply (metis F1-inv-def l1-invariant-def)
apply (rule subsetI)
apply simp
apply (erule bexE)
unfolding new1-post-defs
apply (elim conjE disjE)
apply simp-all
thm Diff-iff l-dom-dom-ar
apply (metis f-in-dom-ar-subsume f-in-dom-ar-the-subsume)

apply (insert l1-input-notempty-def)
apply (insert l1-invariant-def)
unfolding F1-inv-def
apply (erule conjE)+
apply (frule l-disjoint-mapupd-keep-sep[of f1 r s1])
apply assumption+
thm l-munion-apply
apply (subgoal-tac dom f1 ∩ dom [r + s1 ↦ the (f1 r) - s1] = {})
apply simp
apply (simp add: l-munion-dom-ar-assoc l-munion-apply f-in-dom-ar-the-subsume)

  — NOTE: the above simp is VERY slow :-(
apply (split split-if-asm)
apply simp-all
apply (rule-tac x=r in bexI)
apply (smt b-new1-gr-upd-within-req-size l1-input-notempty-def set-mp)
  — NOTE: sledgehammer didn't find this! I just used a previous case that it succeeded and it worked!

apply assumption
apply (frule f-in-dom-ar-subsume)
apply (simp add: l-munion-dom)
by metis

lemma new1-postcondition-state-locs-subset-algebraic:
 PO-new1-postcondition-state-locs-subset r
unfolding PO-new1-postcondition-state-locs-subset-def
apply (intro allI impI)
```

apply (*insert l1-invariant-def*)
unfolding *new1-postcondition-def new1-post-defs*
apply (*elim conjE disjE*)
apply *simp-all*
thm *l-locs-dom-ar-iff*[*of f1 r*]
— expanding dom_ar: depends on nat1_map and disjoint explicitly
l-locs-munion-iff[*of* {*r*}-◁*f1* [*r* + *s1* ↦ *the* (*f1 r*) - *s1*]]
— expanding munion: depends on specialised nat1_map (i.e. dom_ar and singleton) and aux lemma above
— expanding dom_ar: depends on same as above (but for the 2nd goal as well)
l-nat1-map-singleton[*of the*(*f1 r*) - *s1 r*+*s1*]
— Given nat1_iff rule and given assumtpions, just depend on nat1 s1
l-locs-singleton-iff[*of the*(*f1 r*) - *s1 r*+*s1*]
— expanding locs: singleton depends on nat1 s1

apply (*simp add*: *l-locs-dom-ar-iff*
f-F1-inv-disjoint f-F1-inv-nat1-map
Diff-subset)
apply (*subst l-locs-munion-iff*)
apply (*simp add*: *f-F1-inv-nat1-map k-nat1-map-dom-ar-specific*)
apply (*simp add*: *l-nat1-map-singleton*)
apply (*metis f-F1-inv-disjoint f-F1-inv-nat1-map l1-input-notempty-def k-new1-gr-dom-ar-dagger-aux*)
apply (*simp add*: *l-locs-dom-ar-iff*
f-F1-inv-disjoint f-F1-inv-nat1-map
l-locs-singleton-iff
Diff-subset)
by (*metis F1-inv-def b-new1-gr-upd-within-req-size l-locs-of-within-locs subset-trans*)

lemma *new1-postcondition-diff-f-locs-headon*:
PO-new1-postcondition-diff-f-locs r
unfolding *PO-new1-postcondition-diff-f-locs-def*
apply (*intro allI impI*)
unfolding *new1-postcondition-def new1-post-defs*
apply (*elim conjE disjE*)

thm *l-locs-dom-ar-iff*
f-F1-inv-disjoint
l1-invariant-def
apply (*simp add*: *l-locs-dom-ar-iff* — rely on the two inv properties
f-F1-inv-disjoint
f-F1-inv-nat1-map — rely on invariant over f1 not f1'!
l1-invariant-def)
apply (*metis Diff-iff F1-inv-def k-in-locs-iff l1-invariant-def f-dom-locs-of*)

find-theorems - -◁ - ∪*m* -
thm *l-munion-dom-ar-assoc*[*of* {*r*} *f1* [*r* + *s1* ↦ *the* (*f1 r*) - *s1*],*simplified*]
l-disjoint-mapupd-keep-sep[*of f1 r s1*]
l1-input-notempty-def

find-theorems *locs* (- ∪*m* -)
find-theorems *nat1-map* [- ↦ -]
find-theorems *nat1-map* (- -◁ -)
thm *l-locs-munion-iff*[*of* {*r*} -◁ *f1* [*r* + *s1* ↦ *the* (*f1 r*) - *s1*],*simplified*]
thm *l-nat1-map-singleton*[*of the*(*f1 r*) - *s1 r*+*s1*,*simplified*]
k-new1-nat1-map-dom-ar

apply (*insert l1-invariant-def*)
apply (*frule f-F1-inv-disjoint*[*of f1*])
apply (*frule f-F1-inv-nat1-map*[*of f1*])
apply (*insert l1-input-notempty-def*)
apply (*insert l-disjoint-mapupd-keep-sep*[*of f1 r s1*])
find-theorems *simp*:(- ⟹ *False*)
apply (*insert k-new1-nat1-map-dom-ar*[*of r*])
apply (*insert f-in-dom-ar-subsume*[*of r+s1* {*r*} *f1*])
apply (*insert l-nat1-map-singleton*[*of the*(*f1 r*) - *s1 r+s1*,*simplified*])
find-theorems *locs* [- ↦ -]
apply (*simp add*: *l-locs-munion-iff*
atomize-not
l-locs-dom-ar-iff
l-locs-singleton-iff)
find-theorems (- - -) ∪ -
find-theorems - - - = - *name*:*Set*
thm *Un-Diff*
Diff-Un[*of locs f1 locs-of r* (*the* (*f1 r*)) *locs-of* (*r* + *s1*) (*the* (*f1 r*) - *s1*)]

apply (*simp add*: *l-diff-un-not-equal*
l-locs-of-within-locs
b-new1-gr-upd-psubset-req-size)
done

lemma *new1-postcondition-shrinks-f-locs*:
PO-new1-postcondition-shrinks-f-locs r
unfolding *PO-new1-postcondition-shrinks-f-locs-def*
find-theorems - ⊂ -
apply (*intro allI impI*)
apply (*rule psubsetI*)
apply (*metis PO-new1-postcondition-state-locs-subset-def new1-postcondition-state-locs-subset-planned*)
by (*metis PO-new1-postcondition-diff-f-locs-def new1-postcondition-diff-f-locs-headon*)

lemma *new1-postcondition-f-equiv*:
PO-new1-postcondition-f-equiv r
unfolding *PO-new1-postcondition-f-equiv-def new1-postcondition-def new1-post-defs*
apply (*intro allI impI*)
apply (*elim conjE disjE*)
by (*simp-all add*: *Un-absorb l-dom-ar-accum*)

end*"case 2.2.2 [14]: new1-gr*; *above* ≠ *empty*; *below* ≠ *empty* **lemma** *l-new1-dispose-1-identity-case-14*:
0 < *n* ⟹ *r* ∈ *dom f* ⟹ *sep f* ⟹ *Disjoint f* ⟹ *r* + *n* ∉ *dom* ({*r*} -◁ *f*) ⟹
l ∈ *dom* ({*r*} -◁ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) ⟹
l + *the* (({*r*} -◁ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) *l*) = *r* ⟹
f = ({*r* + *n*, *l*}
-◁
(({*r*} -◁ *f*) ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*])
)
∪*m*
[*l* ↦ *the* (({*r*} -◁ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) (*r* + *n*)) + *the* (({*r*} -◁ *f* ∪*m* [*r* + *n* ↦

the (*f r*) - *n*]) *l*) + *n*]
apply (*simp add*: *l-munion-apply l-munion-dom f-in-dom-ar-apply-subsume*)
apply (*cases l=r+n*)
apply (*simp del*: *diff-is-0-eq' diff-is-0-eq*)

apply (*simp add*: *f-in-dom-ar-apply-subsume*)
by (*metis l-dom-ar-not-in-dom sep-def*)

thm *l-munion-dom-ar-assoc*[*of* {*r*} *f* [*r* + *n* ↦ *the* (*f r*) - *n*], *symmetric*,*simplified*]
l-munion-dom-ar-assoc[*of* {*r*+*n*,*l*} (({*r*} -◁ *f*) ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*])
[*l* ↦ *the* (*f r*) - *n* + *the* (({*r*} -◁ *f*) *l*) + *n*],*symmetric*,*simplified*]

lemma *new1-dispose1-identity*:
PO-new1-dispose1-identity-post f n r
unfolding *PO-new1-dispose1-identity-post-def*
apply (*intro allI impI*)
apply (*erule conjE*)
apply (*simp only*: *dispose1-equiv*)
unfolding *new1-post-def dispose1-post2-def*
apply (*elim disjE conjE exE*)+

— case1 [1]: new1_eq
apply (*simp-all*)
apply (*frule k-F1-inv-dom-ar*[*of* - {*r*}])
unfolding *new1-post-eq-def F1-inv-def*
apply (*elim conjE*)
apply (*simp add*: *l-min-loc-dispose1-ext-absorb-above*)
unfolding *dispose1-ext-def*
— case1.1 [2]: new1_eq; below=empty
apply (*cases dispose1-below* ({*r*} -◁ *f*) *r* = *empty*)
thm *l-sum-size-munion-singleton*[*simplified*] *f-d1-not-dispose-above*[*of n r* {*r*} -◁ *f*,*simplified*]
apply (*frule f-d1-not-dispose-above*[*of n r* {*r*} -◁ *f*,*simplified*])
apply (*subst l-sum-size-munion-singleton*)
apply (*metis k-finite-dispose-abovebelow-munion nat1-def*)
apply (*smt Un-commute disjoint-iff-not-equal empty-iff insert-is-Un l-dom-extend l-map-non-empty-dom-conv singleton-iff unionm-in-dom-right*)
apply (*simp add*: *l-munion-empty-lhs l-munion-empty-rhs*
l-min-loc-singleton)

— case 1.1.1 [3]: new1_eq; below=above=empty
apply (*cases dispose1-above* ({*r*} -◁ *f*) *r n* = *empty*)
apply (*simp-all add*: *l-munion-empty-lhs l-dom-ar-none*)
apply (*metis l-munion-subsume*)

— case 1.1.2 [4]: new1_eq; below=empty; not above = empty
apply (*simp add*: *l-dispose1-nonempty-above-singleton l-sum-size-singleton*)
apply (*metis* (*full-types*) *k-empty-dispose-above l-dom-ar-notin-dom-or sep-def*)

— case1.2 [5]: new1_eq; not below = empty
— case 1.2.1 [6]: above=empty; not below = empty
apply (*cases dispose1-above* ({*r*} -◁ *f*) *r n* = *empty*)
apply (*simp add*: *l-dispose1-sep0-below-empty-iff*[*of* {*r*} -◁ *f r n*])
apply (*unfold sep0-def*)
apply *simp*
apply (*erule bexE*)

find-theorems - -◁ - -◁ -
thm *k-min-loc-munion-singleton*[*simplified*]
apply (*simp add*: *l-munion-empty-lhs l-dispose1-below-singleton-useful l-dom-ar-accum*)

apply (*subst k-min-loc-munion-singleton*)
apply (*metis finite-singleton*)
apply (*simp add*: *disjoint-iff-not-equal f-in-dom-ar-notelem*)
apply (*simp*)
apply (*subst l-sum-size-munion-singleton*)
apply (*metis finite-singleton*)
apply (*simp add*: *disjoint-iff-not-equal f-in-dom-ar-notelem*)
apply (*simp add*: *l-sum-size-singleton min-def*)

apply (*metis f-in-dom-ar-apply-subsume l-dom-ar-not-in-dom sep-def*)

— case 1.2.2 [7]: new1_eq; not above = empty; not below = empty
apply (*simp add*: *l-dispose1-sep0-below-empty-iff*[*of* {*r*} -◁ *f r n*])
apply (*unfold sep0-def*)
apply *simp*
apply (*erule bexE*)
find-theorems - -◁ - -◁ -
thm *k-min-loc-munion-singleton*[*simplified*]
apply (*simp add*: *l-dispose1-below-singleton-useful*
l-dispose1-nonempty-above-singleton l-sum-size-singleton)

apply (*subst k-min-loc-munion-singleton*)
apply (*metis finite-singleton*)
apply (*simp add*: *disjoint-iff-not-equal f-in-dom-ar-notelem*)
apply (*simp*)
apply (*subst l-sum-size-munion-singleton*)
— slightly more complicated because there is two munion
apply (*smt k-finite-dispose-abovebelow-munion l-dispose1-below-singleton-useless l-dispose1-nonempty-above-singleton nat1-def*)
apply (*simp add*: *disjoint-iff-not-equal*)
apply (*rule ballI*)
apply (*simp add*: *l-munion-dom*)
apply (*metis sep-def*)

apply (*subst l-sum-size-munion-singleton*)
apply (*metis finite-singleton*)
apply (*simp add*: *disjoint-iff-not-equal*)

apply (*simp add*: *l-sum-size-singleton min-def l-munion-empty-iff*)

apply (*metis f-in-dom-ar-apply-subsume l-dom-ar-not-in-dom sep-def*)

— case 2 [8]: new1_gr
apply (*fold F1-inv-def*)
apply (*fold dispose1-ext-def*)
unfolding *new1-post-gr-def*
apply (*elim conjE*)
apply (*frule k-F1-inv-dom-munion*)
apply (*simp-all* (*no-asm*))
unfolding *F1-inv-def*
apply (*elim conjE*)
apply (*simp add*: *l-min-loc-dispose1-ext-absorb-above*)

unfolding *dispose1-ext-def*

— case2.1 [9]: new1_gr; below=empty
apply (*cases dispose1-below* ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) *r* = *empty*)
apply (*simp add*: *l-munion-empty-rhs l-munion-empty-lhs*)
thm *l-sum-size-munion-singleton*[*simplified*] *f-d1-not-dispose-above*[*of n r* {*r*} -⊲ *f*,*simplified*]
apply (*frule f-d1-not-dispose-above*[*of n r* ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]),*simplified*])
apply (*subst l-sum-size-munion-singleton*)
apply (*metis k-finite-dispose-above*)
apply (*smt Diff-disjoint Int-commute dom-empty dom-fun-upd l-dom-dom-ar l-munion-dom-ar-singleton-subsume option.distinct*(*1*))
apply (*simp add*: *l-munion-empty-lhs l-munion-empty-rhs*
l-min-loc-singleton)

— case 2.1.1 [10]: new1_gr; below=above=empty
apply (*cases dispose1-above* ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) *r n* = *empty*)
apply (*simp-all add*: *l-munion-empty-lhs l-dom-ar-none l-sum-size-singleton*)

thm *l-munion-subsume l-dispose1-sep0-below-empty-iff*[*of* - *r n*]

pr
apply (*simp add*: *l-dispose1-sep0-above-empty-iff*)
unfolding *sep0-def*
apply (*simp*)
apply (*erule notE*)
thm *l-munion-dom*[*of* ({*r*} -⊲ *f*) [*r* + *n* ↦ *the* (*f r*) - *n*]]
apply (*subst l-munion-dom*)
apply *simp-all*
thm *l-disjoint-mapupd-keep-sep k-new1-gr-dom-ar-dagger-aux2*
— apply (metis k_new1_gr_dom_ar_dagger_aux2 nat1_def)
apply (*metis l-disjoint-mapupd-keep-sep l-dom-ar-notin-dom-or nat1-def*)

— case 2.1.2 [11]: new1_gr; below=empty; not above = empty

apply (*simp add*: *l-dispose1-nonempty-above-singleton*)

apply (*simp add*: *l-sum-size-singleton*)

thm *l-munion-apply*[*of* {*r*} -⊲ *f* [*r* + *n* ↦ *the* (*f r*) - *n*],*simplified*]
k-new1-gr-dom-ar-dagger-aux2[*of f r n*]
apply (*simp add*: *l-munion-apply k-new1-gr-dom-ar-dagger-aux2*[*of f r n*])
apply (*insert k-new1-gr-dom-ar-dagger-aux2*[*of f r n*])

apply (*simp add*: *l-dispose1-sep0-above-empty-iff l-dispose1-sep0-below-empty-iff*[*of* - *r n*])
unfolding *sep0-def*
apply (*simp add*: *l-munion-apply*)
apply (*erule-tac x*=*r*+*n* **in** *ballE*,*simp-all*)
apply (*metis l-munion-dom-ar-singleton-subsume l-munion-subsume*)

— case2.2 [12]: new1_gr; not below =empty
— case 2.2.1 [13]: new1_gr; above=empty; not below = empty
apply (*cases dispose1-above* ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) *r n* = *empty*)
apply (*simp add*: *l-dispose1-sep0-above-empty-iff*)
apply (*unfold sep0-def*)

apply (*simp add*: *l-munion-dom*)

— case 2.2.2 [14]: new1_gr; not above = empty; not below = empty
apply (*simp add*: *l-dispose1-sep0-below-empty-iff*[*of* ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) *r n*])
apply (*unfold sep0-def*)
apply *simp*
apply (*erule bexE*)
find-theorems - -⊲ - -⊲ -
thm *k-min-loc-munion-singleton*[*simplified*]
apply (*simp add*: *l-dispose1-below-singleton-useful*
l-dispose1-nonempty-above-singleton
l-sum-size-singleton)

apply (*subst k-min-loc-munion-singleton*)
apply (*metis finite-singleton*)
apply (*simp add*: *disjoint-iff-not-equal*)
apply (*metis sep-def*)
apply (*simp*)
apply (*subst l-sum-size-munion-singleton*)
— slightly more complicated because there is two munion
apply (*smt k-finite-dispose-abovebelow-munion*
l-dispose1-below-singleton-useless
l-dispose1-nonempty-above-singleton nat1-def)
apply (*simp add*: *disjoint-iff-not-equal*)
apply (*rule ballI*)
apply (*simp add*: *l-munion-dom*)
apply (*smt f-in-dom-ar-notelem*)

apply (*subst l-sum-size-munion-singleton*)
apply (*metis finite-singleton*)
apply (*simp add*: *disjoint-iff-not-equal*)

apply (*simp add*: *l-sum-size-singleton min-def l-munion-empty-iff*)

— apply (metis f_in_dom_ar_apply_subsume l_dom_ar_not_in_dom sep_def)
apply (*simp add*: *l-new1-dispose-1-identity-case-14*)
done

lemma *l-new1-dispose-1-identity-case-11*:
0 < *n* ⟹ *r* ∈ *dom f* ⟹ *nat1-map f* ⟹ *Disjoint f* ⟹ *r* + *n* ∈ *dom* ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) ⟹
r + *n* ∉ *dom* ({*r*} -⊲ *f*) ⟹ *f* = {*r* + *n*} -⊲ ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) ∪*m* [*r* ↦ *the* (*f r*)]
by (*metis l-munion-dom-ar-singleton-subsume l-munion-subsume*)

lemma *l-new1-dispose-1-identity-case-11-original*:
0 < *n* ⟹ *r* ∈ *dom f* ⟹ *nat1-map f* ⟹ *Disjoint f* ⟹ *r* + *n* ∈ *dom* ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) ⟹
r + *n* ∉ *dom* ({*r*} -⊲ *f*) ⟹ *f* = {*r* + *n*} -⊲ ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) ∪*m* [*r* ↦ *the* (*f r*)]
find-theorems *simp*:- -⊲ - ∪*m* -
thm *l-munion-dom-ar-assoc*[*of* {*r*+*n*} ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) [*r* ↦ *the* (*f r*)],*simplified*]
l-munion-subsume l-munion-assoc
k-new1-gr-dom-ar-dagger-aux[*of* - *r n*]

apply (*subst l-munion-dom-ar-assoc*[*of* {*r*+*n*} ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) [*r* ↦ *the* (*f r*)],*simplified*])
  apply *simp*
  apply (*simp add*: *l-munion-dom l-dom-dom-ar*)
thm *l-munion-assoc*[*of* {*r*} -⊲ *f* [*r* + *n* ↦ *the* (*f r*) - *n*] [*r* ↦ *the* (*f r*)]]
apply (*subst l-munion-assoc*[*of* {*r*} -⊲ *f* [*r* + *n* ↦ *the* (*f r*) - *n*] [*r* ↦ *the* (*f r*)]],*simp-all*)
thm *l-munion-commute*[*of* [*r* + *n* ↦ *the* (*f r*) - *n*] [*r* ↦ *the* (*f r*)]]
apply (*subst l-munion-commute*[*of* [*r* + *n* ↦ *the* (*f r*) - *n*] [*r* ↦ *the* (*f r*)]],*simp*)
thm *l-munion-assoc*[*of* {*r*} -⊲ *f* [*r* ↦ *the* (*f r*)] [*r* + *n* ↦ *the* (*f r*) - *n*],*symmetric*]
apply (*subst l-munion-assoc*[*of* {*r*} -⊲ *f* [*r* ↦ *the* (*f r*)] [*r* + *n* ↦ *the* (*f r*) - *n*],*symmetric*])
  apply (*simp add*: *l-dom-dom-ar*)
  apply *simp*
apply (*subst l-munion-subsume*[*of r f the*(*f r*),*symmetric*],*simp-all*)
thm *l-munion-subsume*[*of r*+*n f* (*the*(*f r*)) - *n*,*symmetric*]

  l-munion-dom-ar-assoc[*of* {*r*+*n*} *f* [*r* + *n* ↦ *the* (*f r*) - *n*],*simplified*,*symmetric*]

find-theorems *simp*:*dom*(- -⊲ -)
apply (*simp add*: *l-dom-dom-ar*)

thm *b-dagger-munion*[*of f* [*r* + *n* ↦ *the* (*f r*) - *n*],*symmetric*,*simplified*]
  antirestr-then-dagger-notin[*of r*+*n f*]

apply (*simp add*: *l-munion-dom-ar-singleton-subsume*)

done

— case 2.2.1 [13]: new1_gr; above=empty; not below = empty
lemma *l-new1-dispose-1-identity-case-13-original*:
  r + *n* ∉ *dom* ({*r*} -⊲ *f*) ⟹
  r + *n* ∉ *dom* ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) ⟹
  l ∈ *dom* ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) ⟹
  l + *the* (({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) *l*) = *r* ⟹
  f = {*l*} -⊲ ({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) ∪*m* [*l* ↦ *the* (({*r*} -⊲ *f* ∪*m* [*r* + *n* ↦ *the* (*f r*) - *n*]) *l*) + *n*]

by (*simp add*: *l-munion-dom*)

find-theorems *locs* -

thm *l-locs-dom-ar-iff*
  HEAP1SanityProofs.level1-new.new1-postcondition-state-locs-subset-algebraic
  l-disjoint-dispose1-ext
    HEAP1Proofs.level1-dispose.z-F1-inv-dispose1-Disjoint

thm *l-disjoint-singleton-upd*
  HEAP1Proofs.level1-dispose.z-F1-inv-dispose1-Disjoint

find-theorems *locs* - = *locs* -

end

Appendix F

Heap lemmas and proofs (Iain)

theory *HEAP0ProofsIJW*
imports *HEAP0*
begin

theorem (**in** *level0-new*)
  locale0-new-FSB: *PO-new0-feasibility*
unfolding *PO-new0-feasibility-def new0-postcondition-def new0-post-def*
proof -
 from *l0-new0-precondition-def new0-pre-def* **obtain** *f0new r*
   where *f0wit*: *f0new = f0 - locs-of r s0* **and** *isb*: *is-block r s0 f0*
   by *auto*
 moreover have *F0-inv f0new* **using** *l0-invariant-def F0-inv-def f0wit* **by** *simp*
 ultimately show $\exists \cdot f'\ r'.\ (\textit{is-block}\ r'\ \textit{s0}\ \textit{f0} \wedge f' = \textit{f0 - locs-of}\ r'\ \textit{s0}) \wedge \textit{F0-inv}\ f'$ **by** *blast*
qed

theorem (**in** *level0-dispose*)
  locale0-dispose-FSB: *PO-dispose0-feasibility*
unfolding *PO-dispose0-feasibility-def dispose0-postcondition-def dispose0-post-def*
proof -
  from *l0-dispose0-precondition-def dispose0-pre-def* **obtain** *f0new*
    where *f0wit*: $\textit{f0new} = (\textit{f0} \cup \textit{locs-of d0 s0})$ **by** *auto*
  moreover have *F0-inv f0new*
  proof -
   have *finite* (*locs-of d0 s0*) **using** *locs-of-def l0-input-notempty-def* **by** *auto*
   then have *F0-inv* (*f0* $\cup$ *locs-of d0 s0*)
    using *l0-invariant-def F0-inv-def* **by** *simp*
   thus *F0-inv f0new*
    using *f0wit dispose0-post-def* **by** *auto*
  qed
  ultimately show $\exists \cdot f'.\ f' = \textit{f0} \cup \textit{locs-of d0 s0} \wedge \textit{F0-inv}\ f'$ **by** *blast*
qed

end

theory *HEAP1LemmasIJW*
imports *HEAP1*
begin

lemma *invF1-sep-weaken*: *F1-inv f* $\Longrightarrow$ *sep f*
unfolding *F1-inv-def* **by** *simp*

lemma *invF1-Disjoint-weaken*: *F1-inv f* $\Longrightarrow$ *Disjoint f*
unfolding *F1-inv-def* **by** *simp*

lemma *invF1-nat1-map-weaken*: *F1-inv f* $\Longrightarrow$ *nat1-map f*
unfolding *F1-inv-def* **by** *simp*

lemma *invF1-finite-weaken*: *F1-inv f* $\Longrightarrow$ *finite* (*dom f*)
unfolding *F1-inv-def* **by** *simp*

lemma *invF1E*[*elim*!]: *F1-inv f* $\Longrightarrow$ (*sep f* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *nat1-map f* $\Longrightarrow$ *finite* (*dom f*) $\Longrightarrow$ *R*) $\Longrightarrow$ *R*
unfolding *F1-inv-def* **by** *simp*

lemma *invF1I*[*intro*!]: *sep f* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *nat1-map f* $\Longrightarrow$ *finite* (*dom f*) $\Longrightarrow$ *F1-inv f*
unfolding *F1-inv-def* **by** *simp*

lemma *invF1-shape*: *nat1-map f* $\Longrightarrow$ *finite* (*dom f*) $\Longrightarrow$ *VDM-F1-inv f* $\Longrightarrow$ *F1-inv f*
unfolding *F1-inv-def VDM-F1-inv-def* **by** *simp*

lemma *invVDMF1*[*intro*!]: *sep f* $\Longrightarrow$ *Disjoint f* $\Longrightarrow$ *VDM-F1-inv f*
unfolding *VDM-F1-inv-def* **by** *simp*

lemma *ballUnE*[*elim*!]: $\forall \cdot x \in f \cup g.\ P\ x \Longrightarrow (\forall \cdot x \in f.\ P\ x \Longrightarrow \forall \cdot x \in g.\ P\ x \Longrightarrow R) \Longrightarrow R$
by *auto*

lemma *ballUnI*[*intro*!]: $\forall \cdot x \in f.\ P\ x \Longrightarrow \forall \cdot x \in g.\ P\ x \Longrightarrow \forall \cdot x \in f \cup g.\ P\ x$
by *auto*

lemma *setminus-trans*: *X - insert x F* = (*X - F*) - {*x*}
by (*metis Diff-insert*)

lemma *UN-minus*: $\forall \cdot x \in X\text{-}\{y\}.\ P\ x \cap P\ y = \{\} \Longrightarrow (\bigcup\ x \in X\text{-}\{y\}.\ P\ x) = (\bigcup x \in X.\ P\ x)$ - *P y*
by *blast*

lemma *UN-minus-gen*:
$\forall \cdot x \in X.\ \forall \cdot\ y \in\ Y.\ P\ x \cap P\ y = \{\} \Longrightarrow (\bigcup\ x \in X\text{-}Y.\ P\ x) = (\bigcup x \in X.\ P\ x)$ - $(\bigcup y \in Y.\ P\ y)$
by *blast*

lemma *union-comp*: $\{x \in A \cup B.\ P\ x\} = \{x \in A.\ P\ x\} \cup\ \{x \in B.\ P\ x\}$

by *auto*

lemma *nat-min-absorb1*: *min* ((*x*::*nat*) + *y*) *x* = *x*
by *auto*

lemma *not-dom-not-locs-weaken*: *nat1-map f* ⟹ *x* ∉ *locs f* ⟹ *x* ∉ *dom f*
apply (*unfold locs-def*)
apply *simp*
apply (*cases x*∈ *dom f*)
prefer *2*
apply *simp*
apply (*erule-tac x*=*x* **in** *ballE*)
prefer *2*
apply *simp*
apply (*unfold locs-of-def*)
apply (*subgoal-tac nat1* (*the* (*f x*)))
apply *simp*
by (*metis nat1-map-def*)

lemma *k-locs-of-arithI*:
nat1 n ⟹ *nat1 m* ⟹ *a*+*n* ≤ *b* ∨ *b*+*m* ≤ *a* ⟹ *locs-of a n* ∩ *locs-of b m* = {}
unfolding *locs-of-def*
by *auto*

lemma *k-locs-of-arithIff*:
nat1 n ⟹ *nat1 m* ⟹ (*locs-of a n* ∩ *locs-of b m* = {}) = (*a*+*n* ≤ *b* ∨ *b*+*m* ≤ *a*)
unfolding *locs-of-def*
apply *simp*
apply (*rule iffI*)
apply (*erule equalityE*)
apply (*simp-all add*: *disjoint-iff-not-equal*)
apply (*metis* (*full-types*) *add-0-iff le-add1 le-neq-implies-less nat-le-linear not-le*)
by (*metis le-trans not-less*)

lemma *k-locs-of-arithE*:
locs-of a n ∩ *locs-of b m* = {} ⟹ *nat1 m* ⟹ *nat1 n* ⟹ (*a*+*n* ≤ *b* ∨ *b*+*m* ≤ *a* ⟹ *nat1 n* ⟹
nat1 m ⟹ *R*) ⟹ *R*
by (*metis k-locs-of-arithIff*)

lemma *l-locs-of-Locs-of-iff*:
l ∈ *dom f* ⟹ *Locs-of f l* = *locs-of l* (*the* (*f l*))
unfolding *Locs-of-def*
by *simp*

lemma *k-inter-locs-iff*: *nat1 s* ⟹ *nat1-map f* ⟹ (*locs-of x s* ∩ *locs f* = {}) = (∀· *y* ∈ *dom f* .
locs-of x s ∩ *locs-of y* (*the*(*f y*)) = {})
unfolding *locs-def*
by (*smt UNION-empty-conv*(*1*) *inf-SUP*)

lemma *k-in-locs-iff*: *nat1-map f* $\Longrightarrow$ $(x \in locs\ f) = (\exists \cdot y \in dom\ f\ .\ x \in locs\text{-}of\ y\ (the(f\ y)))$
unfolding *locs-def*
by (*metis* (*mono-tags*) *UN-iff*)

lemma *l-locs-of-within-locs*:
nat1-map f $\Longrightarrow$ $x \in dom\ f \Longrightarrow locs\text{-}of\ x\ (the(f\ x)) \subseteq locs\ f$
by (*metis k-in-locs-iff subsetI*)

lemma *b-locs-of-as-set-interval*:
nat1 n $\Longrightarrow$ $locs\text{-}of\ l\ n = \{l..<l+n\}$
unfolding *locs-of-def*
by (*metis Collect-conj-eq atLeastLessThan-def atLeast-def lessThan-def*)

lemma *locs-of-subset*: *nat1* $(m - s) \Longrightarrow locs\text{-}of\ (l + s)\ (m - s) \subseteq locs\text{-}of\ l\ m$
apply (*subst b-locs-of-as-set-interval*, *simp*)
apply (*subst b-locs-of-as-set-interval*,*simp*)
by *simp*

lemma *domf-in-locs*: *nat1-map f* $\Longrightarrow$ $dom\ f \subseteq locs\ f$
unfolding *locs-def*
apply *simp*
by (*metis locs-def not-dom-not-locs-weaken subsetI*)

lemma *locs-of-finite*: *nat1 s* $\Longrightarrow$ *finite* $(locs\text{-}of\ l\ s)$
unfolding *locs-of-def* **by** *auto*

lemma *l-dom-in-locs-of*: *nat1-map f* $\Longrightarrow$ $x \in dom\ f \Longrightarrow\ x \in locs\text{-}of\ x\ (the\ (f\ x))$
apply(*subst b-locs-of-as-set-interval*)
apply (*simp add*: *nat1-map-def*)
apply (*simp add*: *nat1-map-def*)
done

lemma *locs-of-unique*: *nat1 y* $\Longrightarrow$ *nat1 y'* $\Longrightarrow$ $locs\text{-}of\ x\ y = locs\text{-}of\ x'\ y' \Longrightarrow x = x' \wedge y = y'$
apply (*simp add*: *b-locs-of-as-set-interval*)
by (*metis add-left-cancel atLeastLessThan-eq-iff*
comm-monoid-add-class.add.right-neutral nat-add-left-cancel-less)

lemma *locs-of-uniquerange*: *nat1 y* $\Longrightarrow$ *nat1 y'* $\Longrightarrow$ $locs\text{-}of\ x\ y = locs\text{-}of\ x\ y' = (y = y')$
apply (*simp add*: *b-locs-of-as-set-interval*)
by (*metis add-left-cancel atLeastLessThan-eq-iff comm-monoid-add-class.add.left-neutral less-add-eq-less*)

lemma *locs-of-uniquedom*: *nat1-map f* $\Longrightarrow$ *nat1-map g* $\Longrightarrow$ $x \in dom\ f \Longrightarrow x' \in dom\ g \Longrightarrow locs\text{-}of\ x$
$(the\ (f\ x)) = locs\text{-}of\ x'\ (the\ (g\ x')) \Longrightarrow x = x'$
unfolding *nat1-map-def*
apply (*erule-tac x=x* **in** *allE*)
apply (*erule-tac x=x'* **in** *allE*)
apply (*erule impE*) **apply** *simp*
apply (*erule impE*) **apply** *simp*
by (*metis locs-of-unique*)

lemma *locs-of-edge*: x - *1* $\in$ *locs-of a b* $\Longrightarrow$ $x \notin$ *locs-of a b* $\Longrightarrow$ *nat1 b* $\Longrightarrow$ $x = a+b$
by (*auto simp add*: *b-locs-of-as-set-interval*)

lemma *locs-empty*: *locs empty* = {} **unfolding** *locs-def*
by (*metis SUP-empty dom-empty empty-iff nat1-map-def*)

lemma *empty-locs-empty-map*: *nat1-map f* $\Longrightarrow$ *locs f* = {} $\Longrightarrow$ *f* = *empty*
unfolding *locs-def* **apply** *simp*
by (*metis domIff empty-iff l-dom-in-locs-of*)

lemma *locs-of-pred*: $x \neq a$ $\Longrightarrow$ *nat1 b* $\Longrightarrow$ $x \in$ *locs-of a b*
$\Longrightarrow$ x - *1* $\in$ *locs-of a b*
apply (*simp add*: *b-locs-of-as-set-interval*) **by** *auto*

lemma *locs-of-pred2*: **assumes** *xgr0*: $x>0$ **and** *nat1f*: *nat1-map f*
and *minusone*: x - *1* $\in$ *locs-of a* (*the* (*f a*))
and *xindom*: $x \in$ *dom f* **and** *aindom*: $a \in$ *dom f*
and *Disj*: *Disjoint f*
shows $x \notin$ *locs-of a* (*the* (*f a*))
proof -
have $x \in$ *locs-of x* (*the* (*f x*)) **by** (*metis l-dom-in-locs-of nat1f xindom*)
from *Disj* **have** *locs-of x* (*the* (*f x*)) $\cap$ *locs-of a* (*the* (*f a*)) = {}
unfolding *Disjoint-def disjoint-def Locs-of-def* **apply** *simp*
apply (*erule-tac* $x = x$ **in** *ballE*)
apply (*erule-tac* $x = a$ **in** *ballE*)
apply (*erule impE*)
apply (*rule notI*)
proof -
assume $x=a$
then have $*$: x - *1* $\in$ *locs-of x* (*the* (*f x*)) **by** (*metis minusone*)
have $**$: x - *1* $\notin$ *locs-of x* (*the* (*f x*))
apply (*subst b-locs-of-as-set-interval*)
apply (*metis nat1-map-def nat1f xindom*)
using *xgr0* **by** *auto*
from $*$ $**$ **show** *False* **by** *auto*
next
assume *locs-of x* (*the* (*f x*)) $\cap$ *locs-of a* (*the* (*f a*)) = {}
then show *locs-of x* (*the* (*f x*)) $\cap$ *locs-of a* (*the* (*f a*)) = {}
by *simp*
next
assume $a \notin$ *dom f*
then show *locs-of x* (*the* (*f x*)) $\cap$ *locs-of a* (*the* (*f a*)) = {} **using** *aindom* **by** *auto*
next
assume $x \notin$ *dom f* **then show** *locs-of x* (*the* (*f x*)) $\cap$ *locs-of a* (*the* (*f a*)) = {}
using *xindom* **by** *simp*
qed
then show $x \notin$ *locs-of a* (*the* (*f a*))
by (*metis* ‹$x \in$ *locs-of x* (*the* (*f x*))› *disjoint-iff-not-equal*)
qed

lemma *locs-of-extended*: $\exists \cdot y \in$ *locs-of x a*. $y \notin$ *locs-of x b* $\Longrightarrow$ *nat1 a* $\Longrightarrow$ *nat1 b* $\Longrightarrow$ $a > b$
apply (*erule bexE*)
by (*simp add*: *b-locs-of-as-set-interval*)

```
lemma l-plus-s-not-in-f:
assumes  inv: F1-inv f and lindom: l ∈ dom f
  and f1biggers: the(f l) > sand nat1s: nat1 s
  shows l+s ∉ dom f
proof
  assume lsindom: l + s ∈ dom f
  then obtain y where the (f (l+s)) = y by auto
   have ∗: nat1 (the(f(l+s))) by (metis inv invF1-nat1-map-weaken lsindom nat1-map-def)
  from f1biggers have l+ the(f l) > l+s by auto
  from inv have inlocs:l+s ∈ locs-of l (the(f l))
  proof
   have nat1 (the(f l)) by (metis inv invF1-nat1-map-weaken lindom nat1-map-def)
   then show ?thesis
    unfolding locs-of-def
    by (simp add: f1biggers)
  qed
  have notl: l+s ≠ l using nat1s by auto
  have notinlocs: l+s ∉ locs-of l (the(f l))
  proof -
    have locs-of (l+s) (the(f(l+s))) ∩   locs-of l (the(f l)) = {}
      by (metis (full-types) Disjoint-def F1-inv-def Locs-of-def
        disjoint-def inv lindom lsindom notl)
      moreover have l+s ∈ locs-of (l+s) (the(f(l+s)))
        unfolding locs-of-def using ∗ by simp
      ultimately show ?thesis  by auto
    qed
    from inlocs notinlocs show False by auto
qed

lemma top-locs-of: nat1 y ⟹ x + y - 1 ∈ locs-of x y
unfolding locs-of-def
by simp

lemma  top-locs-of2: (the (f l)) > s ⟹ nat1 s ⟹  l + s - 1 ∈ locs-of l (the (f l))
unfolding locs-of-def
  by auto

lemma minor-sep-prop: x ∈ dom f ⟹ l ∈ dom f⟹ l<x ⟹ F1-inv f⟹  l + the (f l) ≤ x
apply(erule invF1E)
 apply (unfold Disjoint-def)
apply(erule-tac x=x in ballE)
apply(erule-tac x=l in ballE)
apply (erule impE)
apply simp
apply (unfold disjoint-def)
apply (unfold Locs-of-def)
apply simp
apply (erule k-locs-of-arithE)
apply (metis nat1-map-def)
apply (metis nat1-map-def)
apply (metis add-leE not-less)
apply metis
by metis
```

theorem *locs-unique*:
assumes *locs-eq*: $locs\ f = locs\ g$
and *invf*: *F1-inv f*
and *invg*: *F1-inv g*
and *notempf*: $f \neq empty$ **and** *notempg*: $g \neq empty$
shows $f = g$
proof -
have *dom-eq*: $dom\ f = dom\ g$
proof (*rule ccontr*)

assume *doms-not-equal*: $dom\ f \neq dom\ g$
have *elem-in-fnotg-or-gnotf*: $(\exists\cdot\ x \in dom\ f.\ x \notin dom\ g) \vee (\exists\cdot\ x \in dom\ g.\ x \notin dom\ f)$
by (*metis* (*full-types*) *doms-not-equal subsetI subset-antisym*)
then show *False*
proof(*elim bexE disjE*)
fix *x*
assume *xinf*: $x \in dom\ f$ **and** *xnoting*: $x \notin dom\ g$
show *False*
proof (*cases x>0*)
assume *xgr0*: *x>0*
have $x \in locs\text{-}of\ x\ (the\ (f\ x))$
by (*metis invF1-nat1-map-weaken invf l-dom-in-locs-of xinf*)
then have $x \in locs\ f$
by (*metis invF1-nat1-map-weaken invf not-dom-not-locs-weaken xinf*)
then have $x \in locs\ g$
by (*metis locs-eq*)
then have $\exists\cdot y \in dom\ g.\ x \in locs\text{-}of\ y\ (the\ (g\ y))$
by (*metis invF1-nat1-map-weaken invg k-in-locs-iff*)
then obtain *y* **where** *ying*: $y \in dom\ g$ **and** *xlocsofy*: $x \in locs\text{-}of\ y\ (the\ (g\ y))$
by *auto*
from *ying xlocsofy* **have** $x \neq y$ **by** (*metis xnoting*)
then have $x - 1 \in locs\text{-}of\ y\ (the\ (g\ y))$
by (*metis invF1-nat1-map-weaken invg locs-of-pred nat1-map-def xlocsofy ying*)
then have $x - 1 \in locs\ g$ **by** (*metis invF1-nat1-map-weaken invg k-in-locs-iff ying*)
then have *xminus1-in-locsf*: $x - 1 \in locs\ f$ **by** (*metis locs-eq*)

from *invf* **have** *sepf*: *sep f* **by** (*rule invF1-sep-weaken*)
from *invf* **have** *Disjf*: *Disjoint f* **by** (*rule invF1-Disjoint-weaken*)
have $x - 1 \notin locs\ f$
proof
let $?x' = x - 1$
assume *xminusinlocs*: $?x' \in locs\ f$
then have $\exists\cdot\ below \in dom\ f.\ ?x' \in locs\text{-}of\ below\ (the\ (f\ below))$
by (*metis invF1-nat1-map-weaken invf k-in-locs-iff*)
then obtain *below* **where** *belowinf*: $below \in dom\ f$
and *locsofbelow*: $?x' \in locs\text{-}of\ below\ (the\ (f\ below))$
by *auto*
have $x \in dom\ f$ **by** (*metis xinf*)
have *notlocsofx*: $x \notin locs\text{-}of\ below\ (the\ (f\ below))$
by (*metis invF1E belowinf invf locs-of-pred2 locsofbelow xgr0 xinf*)
from *locsofbelow notlocsofx* **have** $x = below + the\ (f\ below)$
by (*metis belowinf comm-monoid-diff-class.diff-cancel le-add-diff-inverse*
less-nat-zero-code linorder-neqE-nat locs-of-edge nat1-def order-refl sep-def sepf)

```
      thus False by (metis belowinf sep-def sepf xinf)
     qed
     thus False by (metis xminus1-in-locsf)
    next
     assume ¬ x >0
     then have xzero: x = 0 by (metis neq0-conv)
     have x ∈ locs-of x (the (f x))
        by (metis invF1-nat1-map-weaken invf l-dom-in-locs-of xinf)
     then have x∈ locs f
      by (metis invF1-nat1-map-weaken invf not-dom-not-locs-weaken xinf)
     then have x ∈ locs g by (metis locs-eq)
      then have ∃·y ∈ dom g. x∈ locs-of y (the (g y)) by (metis invF1-nat1-map-weaken invg
k-in-locs-iff)
     then obtain y where ying: y∈ dom g and xlocsofy: x∈ locs-of y (the (g y))
     by auto
     have ynoteqx: y ≠ x by (metis xnoting ying)

     have locs-of y (the (g y)) = {y..<y + (the (g y))}
      by (metis b-locs-of-as-set-interval invF1-nat1-map-weaken invg nat1-map-def ying)
     then have x ∈ {y..<y + (the (g y))} by (metis xlocsofy)
     then have xeqy: x = y using xzero by auto
     from ynoteqx and xeqy show False by simp
    qed
   next
    fix x
    assume xing: x ∈ dom g
    and xnotinf: x∉ dom f
    show False
    proof (cases x>0)
     assume xgr0: x>0
     have x ∈ locs-of x (the (g x))
     by (metis invF1-nat1-map-weaken xing invg l-dom-in-locs-of)
     then have x∈ locs g
      by (metis invF1-nat1-map-weaken invg not-dom-not-locs-weaken xing)
     then have x ∈ locs f by (metis locs-eq)
      then have ∃·y ∈ dom f. x∈ locs-of y (the (f y)) by (metis invF1-nat1-map-weaken invf
k-in-locs-iff)
     then obtain y where ying: y∈ dom f and xlocsofy: x∈ locs-of y (the (f y))
      by auto
     from ying xlocsofy have x ≠ y by (metis xnotinf)
     then have x - 1 ∈ locs-of y (the (f y))
      by (metis invF1-nat1-map-weaken invf locs-of-pred nat1-map-def xlocsofy ying)
     then have x - 1 ∈ locs f by (metis invF1-nat1-map-weaken invf k-in-locs-iff ying)
     then have xminus1ing: x - 1 ∈ locs g by (metis locs-eq)

     from invg have sepg: sep g by (rule invF1-sep-weaken)
     from invg have Disjg: Disjoint g by (rule invF1-Disjoint-weaken)
     have x - 1 ∉ locs g
     proof
      let ?x' = x - 1
      assume xminusinlocs: ?x' ∈ locs g
      then have ∃· below ∈ dom g. ?x' ∈ locs-of below (the (g below))
        by (metis invF1-nat1-map-weaken invg k-in-locs-iff)
      then obtain below where belowing: below∈ dom g
           and locsofbelow: ?x' ∈ locs-of below (the (g below))
       by auto
```

```
        have x∈ dom g by (metis xing)
        have notlocsofx: x ∉ locs-of below (the (g below))
         by (metis Disjg HEAP1LemmasIJW.invF1-nat1-map-weaken
             belowing invg locs-of-pred2 locsofbelow xgr0 xing)
        from locsofbelow notlocsofx have x = below + the (g below)
         by (metis invF1-nat1-map-weaken belowing invg locs-of-edge nat1-map-def)
        thus False by (metis belowing sep-def sepg xing)
       qed
       thus False by (metis xminus1ing)
      next
       assume ¬ x >0
       then have xzero: x = 0 by (metis neq0-conv)
       have x ∈ locs-of x (the (g x))
        by (metis invF1-nat1-map-weaken invg l-dom-in-locs-of xing)
       then have x∈ locs g
        by (metis invF1-nat1-map-weaken invg not-dom-not-locs-weaken xing)
       then have x ∈ locs f by (metis locs-eq)
        then have ∃·y ∈ dom f. x∈ locs-of y (the (f y)) by (metis invF1-nat1-map-weaken invf
k-in-locs-iff)
       then obtain y where yinf: y∈ dom f and xlocsofy: x∈ locs-of y (the (f y))
        by auto
       have ynotx: y ≠ x by (metis xnotinf yinf)

       have  locs-of y (the (f y)) =  {y..<y + (the (f y))}
        by (metis b-locs-of-as-set-interval invF1-nat1-map-weaken invf nat1-map-def yinf)
       then have x ∈ {y..<y + (the (f y))} by (metis xlocsofy)
       then have xeqy: x = y using xzero by auto
       from ynotx and xeqy show False by simp
     qed
   qed
  qed
  show ?thesis
  proof
   fix x
   show f x = g x
   proof (cases x ∈ dom f)
    assume xinf: x ∈ dom f
    show ?thesis
    proof (cases x ∈ dom g)
     assume xing: x∈ dom g
     have (the ( f x)) = (the (g x))
     proof -
      have nat1fx:  nat1 (the ( f x)) by (metis invF1-nat1-map-weaken dom-eq invf nat1-map-def
xing)
      have nat1gx: nat1 (the ( g x)) by (metis invF1-nat1-map-weaken dom-eq invg nat1-map-def
xing)
      have locs-of x (the (f x)) = locs-of x (the (g x))
      proof(rule ccontr)
       assume locs-of-f-not-g:locs-of x (the (f x)) ≠ locs-of x (the (g x))
       then have (∃· y ∈ locs-of x (the (f x)). y ∉  locs-of x (the (g x)))
                 ∨ (∃· y ∈ locs-of x (the (g x)). y ∉  locs-of x (the (f x)))
         by auto
       from this show False
       proof
        assume ∃·y∈locs-of x (the (f x)). y ∉ locs-of x (the (g x))
        then have fgrg: the (f x) > the (g x)
```

```
            by (metis locs-of-extended nat1fx nat1gx)
  have firstpartcontr: x + the (g x) ∉ dom g
            by (metis invF1-sep-weaken invg sep-def xing)
          then have x + the (g x) ∈ locs-of x (the (f x))
            by (metis b-locs-of-as-set-interval nat1fx nat1gx  locs-of-f-not-g
              fgrg locs-of-edge top-locs-of2)
          then have x + the (g x) ∈ locs f
            by (metis invF1-nat1-map-weaken invf k-in-locs-iff xinf)
          then have  x + the (g x) ∈ locs g by (metis locs-eq)
          then have ∃·loc ∈ dom g . x + the (g x) ∈ locs-of loc (the (g loc))
            by (metis invF1-nat1-map-weaken invg k-in-locs-iff)
          then obtain loc where locing: loc ∈ dom g and x+ the (g x) ∈ locs-of loc (the (g loc))
            by auto
          have x + the (g x) - 1 ∈ locs-of x (the (g x))
            by (metis nat1gx top-locs-of)
          have locnotg: loc ≠ x
          proof
            assume loceqx: loc = x
            then have loc+ the (g loc) ∈ locs-of loc (the (g loc))
              by (metis ‹x + the (g x) ∈ locs-of loc (the (g loc))›)
            moreover have loc+ the (g loc) ∉ locs-of loc (the (g loc))
              using b-locs-of-as-set-interval by (simp del: nat1-def add: nat1-map-def nat1gx loceqx)
            ultimately show False by simp
          qed
          from invg have Disjoint g by (rule invF1-Disjoint-weaken)
          then have  locs-of loc (the (g loc)) ∩  locs-of x (the (g x)) = {}
           unfolding Disjoint-def Locs-of-def
           apply (simp add: locing)
           apply (erule-tac x=loc in ballE)
           apply (erule-tac x=x in ballE)
           apply (erule impE)
           apply (rule locnotg)
           apply (metis disjoint-def)
           apply (simp add: xing)
           by (simp add: locing)
          have loc = x + the (g x)

           by (metis (hide-lams, full-types) F1-inv-def ‹x + the (g x) - 1 ∈ locs-of x (the (g x))›
            ‹x + the (g x) ∈ locs-of loc (the (g loc))› ‹locs-of loc (the (g loc)) ∩ locs-of x (the (g x))
= {}›
                comm-monoid-add-class.add.right-neutral disjoint-iff-not-equal dom-eq inf.commute
invg locing
                locs-of-pred nat1-def neq0-conv sep-def)
         then have x + the (g x) ∈ dom g by (metis locing)
         thus False using firstpartcontr by auto
        next
         assume ∃·y∈locs-of x (the (g x)). y ∉ locs-of x (the (f x))
        then have ggrf: the (g x) > the (f x) by (metis (full-types) locs-of-extended nat1fx nat1gx)
  have firstpartcontr: x + the (f x) ∉ dom f
            by (metis invF1-sep-weaken dom-eq invf sep-def xing)
          then have x + the (f x) ∈ locs-of x (the (g x))
            by (metis ggrf b-locs-of-as-set-interval locs-of-edge
               locs-of-f-not-g nat1fx nat1gx top-locs-of2)
          then have x + the (f x) ∈ locs g
           by (metis invF1-nat1-map-weaken invg k-in-locs-iff xing)
          then have  x + the (f x) ∈ locs f by (metis locs-eq)
```

```
            then have ∃·loc ∈ dom f . x + the (f x) ∈ locs-of loc (the (f loc))
              by (metis invF1-nat1-map-weaken invf k-in-locs-iff)
            then obtain loc where locinf: loc ∈ dom f and x+ the (f x) ∈ locs-of loc (the (f loc))
             by auto
            have x + the (f x) - 1 ∈ locs-of x (the (f x))
             by (metis nat1fx top-locs-of)
            have locnotg: loc ≠ x
            proof
              assume loceqx: loc = x
              then have loc+ the (f loc) ∈ locs-of loc (the (f loc))
               by (metis ‹x + the (f x) ∈ locs-of loc (the (f loc))›)
              moreover have loc+ the (f loc) ∉ locs-of loc (the (f loc))
               by (simp del: nat1-def add:  b-locs-of-as-set-interval nat1fx loceqx)
              ultimately show False by simp
            qed
            from invf have Disjoint f by (metis invF1-Disjoint-weaken)
            then have  locs-of loc (the (f loc)) ∩  locs-of x (the (f x)) = {}
             unfolding Disjoint-def Locs-of-def
             apply (simp add: locinf)
             apply (erule-tac x=loc in ballE)
             apply (erule-tac x=x in ballE)
             apply (erule impE)
             apply (rule locnotg)
             apply (metis disjoint-def)
             apply (simp add: xinf)
             by (simp add: locinf)
            have loc = x + the (f x)
             by (metis (hide-lams, full-types) F1-inv-def ‹x + the (f x) - 1 ∈ locs-of x (the (f x))›
                ‹x + the (f x) ∈ locs-of loc (the (f loc))› ‹locs-of loc (the (f loc)) ∩ locs-of x (the (f x))
= {}›
                comm-monoid-add-class.add.right-neutral disjoint-iff-not-equal dom-eq inf.commute invf
locinf
                locs-of-pred nat1-def neq0-conv sep-def)
            then have x + the (f x) ∈ dom f by (metis locinf)
            thus False using firstpartcontr by auto
          qed
         qed
         then show ?thesis by (metis (full-types) locs-of-unique nat1fx nat1gx)
       qed
       thus ?thesis
        using xinf xing by auto
      next
       assume notg: x ∉ dom g then have notf:x ∉ dom f
         using dom-eq by simp
       from notg notf show ?thesis by auto
     qed
    next
     assume xnotf: x ∉ dom f
     then have x∉ dom g using dom-eq by simp
     thus ?thesis using xnotf by auto
   qed
 qed
qed

lemma locs-singleton:
    assumes ∗: nat1 y
```

```
    shows locs [x ↦ y] = locs-of x y
proof -
  from ∗ have nat1-map [x ↦ y]
    by (metis dom-empty empty-iff fun-upd-same l-inmapupd-dom-iff nat1-map-def the.simps)
  then show ?thesis unfolding locs-def by simp
qed


lemma locs-of-subset-range: x > 0 ⟹ y > 0 ⟹ locs-of l x ⊆ locs-of l y ⟹ y ≥ x
 by(simp add: b-locs-of-as-set-interval)

lemma locs-of-subset-range-gr:
    shows x>0 ⟹ y > 0 ⟹ l > l' ⟹ locs-of l x ⊆ locs-of l' y ⟹ y ≥ x
by (simp add: b-locs-of-as-set-interval)

lemma locs-of-subset-top-bottom: b > 0 ⟹ y > 0 ⟹ a ∈ locs-of x y ⟹ a + b - 1 ∈ locs-of x y
⟹ locs-of a b ⊆ locs-of x y
apply (simp add: b-locs-of-as-set-interval)
by auto

lemma less-a-not-in-locs-of: b>0 ⟹ a > l ⟹ l ∉ locs-of a b
  apply (subst b-locs-of-as-set-interval)
  apply simp
  by simp

lemma edge-not-in-locs-of: b>0 ⟹ a+b ∉ locs-of a b
  apply (subst b-locs-of-as-set-interval)
  apply simp
  by simp


lemma after-locs-of-not-in-locs: assumes invf: F1-inv f1
  and mindom: m∈ dom f1
shows m + (the (f1 m)) ∉ locs f1
proof
  assume  m + the (f1 m) ∈ locs f1
  then have ∃·n ∈ dom f1.  m + the (f1 m) ∈ locs-of n (the (f1 n))
  by (metis invF1-nat1-map-weaken invf k-in-locs-iff)
  then obtain n where nindom: n∈ dom f1 and
         locsofn: m + the (f1 m) ∈ locs-of n (the (f1 n))
           by auto
  have  m + the (f1 m) ∉ locs-of m (the (f1 m))
 apply (rule edge-not-in-locs-of) by (metis invF1-sep-weaken comm-monoid-add-class.add.right-neutral

                                    invf mindom neq0-conv sep-def)
  then have m ≠ n by (metis locsofn)
  have sep f1 by (metis invF1-sep-weaken invf)
  then have  m + the (f1 m) ∉ dom f1 by (metis mindom sep-def)
  moreover have m + the (f1 m) ∈ dom f1
  proof (rule ccontr)
    assume m + the (f1 m) ∉ dom f1
   have m + the (f1 m) ∈ locs-of n (the (f1 n)) by (metis locsofn)
   then have m + the (f1 m) - 1 ∈ locs-of n (the (f1 n))
     by (metis invF1-nat1-map-weaken ‹m + the (f1 m) ∉ dom f1› invf locs-of-pred nat1-map-def
nindom)
   moreover have m + the (f1 m) - 1 ∈ locs-of m (the (f1 m))
```

```
    by (metis invF1-nat1-map-weaken invf mindom nat1-map-def top-locs-of)
  moreover have Disjoint f1 by (metis invF1-Disjoint-weaken invf)
  moreover  have locs-of n (the (f1 n)) ∩ locs-of m (the (f1 m)) = {}
      by (metis Disjoint-def ⟨m ≠ n⟩ calculation(3) disjoint-def
            l-locs-of-Locs-of-iff mindom nindom)
  ultimately show False by auto
  qed
  ultimately show False by auto
qed

lemma locs-of-boundaries: b > 0 ⟹ l∈ locs-of a b ⟹ l≥ a ∧ l < a+b
   by (simp add: b-locs-of-as-set-interval)

lemma locs-locs-of-subset:
  assumes subset: locs-of l s1 ⊆ locs f1
  and invf: F1-inv f1
  and nat1s1: nat1 s1
  shows ∃·m ∈ dom f1. locs-of l s1 ⊆ locs-of m (the (f1 m))
proof -
  have l ∈ locs-of l s1 using nat1s1
    by (simp add: b-locs-of-as-set-interval)
   then  have l ∈ locs f1 using subset by auto
   then have l ∈(⋃ s∈dom f1. locs-of s (the (f1 s)))
    unfolding locs-def Locs-of-def
     by (simp add: invf invF1-nat1-map-weaken)
    have ∃· m∈ dom f1. l ∈ locs-of m (the (f1 m))
     by (metis invF1-nat1-map-weaken ⟨l ∈ locs f1⟩ invf k-in-locs-iff)
   then obtain m where mindom: m∈dom f1 and
            linlocsof: l ∈ locs-of m (the (f1 m))
     by auto

   have l+s1 - 1 ∈ locs-of l s1
     by (metis nat1s1 top-locs-of)
   then have l+s1 - 1 ∈ locs f1
     by (metis set-mp subset)
   then have ∃· n∈ dom f1. l+s1 - 1 ∈ locs-of n (the (f1 n))
     by (metis invF1-nat1-map-weaken invf k-in-locs-iff)
   then obtain n where nindom: n∈dom f1 and
          lplusinlocsof: l+s1 - 1  ∈ locs-of n (the (f1 n))
     by auto

   have meqn: m = n
   proof (rule ccontr)
     assume noteq: m≠n
     then have m + (the (f1 m)) ∈ locs-of l s1
     proof -
       have m ≤ l by (metis invF1-sep-weaken comm-monoid-add-class.add.right-neutral invf linlocsof
locs-of-boundaries mindom neq0-conv sep-def)
     moreover have l < m + the (f1 m) by (metis invF1-nat1-map-weaken invf linlocsof locs-of-boundaries
mindom nat1-def nat1-map-def)
     moreover have n ≤ l + s1 - 1 by (metis invF1-sep-weaken comm-monoid-add-class.add.right-neutral
invf locs-of-boundaries lplusinlocsof neq0-conv nindom sep-def)
     moreover have l+s1 - 1 < n + the (f1 n) by (metis invF1-sep-weaken comm-monoid-add-class.add.right-neutral
invf locs-of-boundaries lplusinlocsof neq0-conv nindom sep-def)
       moreover have m+the(f1 m) ≤ n
       proof(rule ccontr)
```

```
      assume ∗: ¬ m + the (f1 m) ≤ n
      then have ∗∗: n < m + the (f1 m) by (metis not-less)
      moreover have n ≥ m
        by (smt ⟨l + s1 - 1 < n + the (f1 n)⟩ ⟨m ≤ l⟩ invf mindom minor-sep-prop nat1-def nat1s1
nindom)
      moreover have ∗∗∗: n ∈ locs-of m (the (f1 m))
        by (metis ∗ calculation(2) invf mindom minor-sep-prop neq-le-trans nindom noteq)
      moreover have Disjoint f1 by (metis invF1-Disjoint-weaken invf)
      moreover have locs-of n (the (f1 n)) ∩ locs-of m (the (f1 m)) = {}
       by (smt ∗ ∗∗∗ ⟨l < m + the (f1 m)⟩ ⟨m ≤ l⟩ invf less-a-not-in-locs-of mindom minor-sep-prop
nindom noteq)
      moreover have n ∈ locs-of n (the (f1 n))
         by (metis invF1-nat1-map-weaken invf l-dom-in-locs-of nindom)
      ultimately show False
        by auto
     qed
     ultimately show ?thesis by (auto simp: b-locs-of-as-set-interval nat1s1)
   qed
   moreover have m + (the (f1 m)) ∉ locs f1
      by (metis after-locs-of-not-in-locs invf mindom)
   ultimately show False by (metis in-mono subset)
  qed
  have locs-of l s1 ⊆ locs-of m (the (f1 m))
  proof (rule locs-of-subset-top-bottom)
   show 0 <s1 by (metis nat1-def nat1s1)
  next
   show  0 < the (f1 m) by (metis invF1-sep-weaken invf mindom
            monoid-add-class.add.right-neutral neq0-conv sep-def)
  next
   show l ∈ locs-of m (the (f1 m))  by (rule linlocsof)
  next
   show  l + s1 - 1 ∈ locs-of m (the (f1 m)) by (metis lplusinlocsof meqn)
  qed
  thus ?thesis by (metis meqn nindom)
qed
```

```
lemma nat1-map-empty: nat1-map empty
   by (metis dom-empty empty-iff nat1-map-def)
```

```
lemma dom-ar-nat1-map:
  assumes ∗: nat1-map f
  shows nat1-map  (s -◁ f)
unfolding nat1-map-def dom-antirestr-def
using ∗ nat1-map-def by (simp add: domIff)
```

lemma *dagger-nat1-map*:
nat1-map f $\Longrightarrow$ *nat1-map g* $\Longrightarrow$ *nat1-map* (*f* † *g*)
unfolding *nat1-map-def dagger-def* **by** (*metis* (*full-types*) *Un-iff dom-map-add map-add-dom-app-simps*(*1*) *map-add-dom-app-simps*(*3*))

lemma *unionm-nat1-map*:
dom f ∩ *dom g* = {} $\Longrightarrow$ *nat1-map f* $\Longrightarrow$ *nat1-map g* $\Longrightarrow$ *nat1-map* (*f* ∪*m g*)
unfolding *munion-def*
by (*simp add*: *dagger-nat1-map*)

lemma *unionm-singleton-nat1-map*:
assumes *disjdom*: *a* ∉ *dom f*
and *nat1f*: *nat1-map f*
and *nat1b*: *nat1 b*
shows *nat1-map* (*f* ∪*m* [*a* ↦ *b*])
proof (*rule unionm-nat1-map*)
show *nat1-map f* **by** (*rule nat1f*)
next
show *nat1-map* [*a* ↦ *b*]
using *nat1b* **by** (*simp add*: *nat1-map-def*)
next
show *dom f* ∩ *dom* [*a* ↦ *b*] = {}
using *disjdom* **by** *simp*
qed

lemma *locs-of-sum-range*: *nat1 y* $\Longrightarrow$ *nat1 z* $\Longrightarrow$ *locs-of x* (*y*+*z*) = (*locs-of x y*) ∪ (*locs-of* (*x*+*y*) *z*)
apply (*subst b-locs-of-as-set-interval*)
apply *simp*
apply (*subst b-locs-of-as-set-interval*, *simp*)
apply (*subst b-locs-of-as-set-interval*, *simp*)
by *auto*

lemma *dom-ar-finite*:
assumes ∗: *finite* (*dom f*)
shows *finite* (*dom* (*s* -◁ *f*))
proof(*rule finite-subset*)
show *dom* (*s* -◁ *f*) ⊆ *dom f* **by** (*rule f-dom-ar-subset-dom*)
show *finite* (*dom f*) **by** (*rule* ∗)
qed

lemma *dagger-finite*:
assumes ∗: *finite* (*dom f*) *finite* (*dom g*)
shows *finite* (*dom* (*f* † *g*))
by (*simp add*: *l-dagger-dom* ∗)

lemma *dagger-singleton-finite*:
assumes ∗: *finite* (*dom f*)

shows *finite* (*dom* (*f* † [*a* ↦ *b*]))
by (*simp add*: *l-dagger-dom* ∗)

lemma *unionm-finite*:
assumes *disjdom*: *dom f* ∩ *dom g* = {} **and** ∗: *finite* (*dom f*) *finite* (*dom g*)
shows *finite* (*dom* (*f* ∪*m g*))
by (*metis* ∗ *l-dagger-dom disjdom finite-UnI munion-def*)

lemma *unionm-singleton-finite*:
assumes *disjdom*: *a* ∉ *dom f*
and ∗: *finite* (*dom f*)
shows *finite* (*dom* (*f* ∪*m* [*a* ↦ *b*]))
by (*simp add*: *unionm-finite* ∗ *disjdom*)

lemma *dom-ar-sep*:
assumes ∗:*sep f*
shows *sep* (*s* -◁ *f*)
by (*smt* ∗ *f-in-dom-ar-subsume sep-def f-in-dom-ar-the-subsume*)

lemma *singleton-sep*: *nat1 b* ⟹ *sep* [*a* ↦ *b*]
unfolding *sep-def* **by** *simp*

lemma *dagger-singleton-sep*:
assumes ∗: *sep f*
and ∗∗∗: ∀·*l*∈ *dom f*. *l*+*the* (*f l*) ∉ *dom* ([*a* ↦ *b*])
and ∗∗∗∗: *a*+*b* ∉ *dom f*
and *anotinf*: *a* ∉ *dom f*
and *nat1b*: *nat1 b*
shows *sep* (*f* † [*a* ↦ *b*])
unfolding *sep-def*
proof(*subst l-dagger-dom*, *rule ballUnI*)
show ∀·*l*∈*dom f*. *l* + *the* ((*f* † [*a* ↦ *b*]) *l*) ∉ *dom* (*f* † [*a* ↦ *b*])
by (*metis* ∗ ∗∗∗ *anotinf dagger-def domIff fun-upd-apply map-add-empty map-add-upd sep-def*)
next
show ∀·*l*∈*dom* [*a* ↦ *b*]. *l* + *the* ((*f* † [*a* ↦ *b*]) *l*) ∉ *dom* (*f* † [*a* ↦ *b*])
by (*smt singleton-sep nat1b* ∗∗∗∗ *dagger-def domIff fun-upd-same l-inmapupd-dom-iff*
map-add-None map-add-dom-app-simps(*1*) *sep-def the.simps*)
qed

lemma *unionm-singleton-sep*:
assumes *disjoint-dom*: *a* ∉ *dom f*
and ∗: *sep f*
and ∗∗∗: ∀·*l*∈ *dom f*. *l*+*the* (*f l*) ∉ *dom* ([*a* ↦ *b*])
and ∗∗∗∗: *a*+*b* ∉ *dom f*

and *nat1b*: *nat1 b*
shows *sep* (*f* ∪*m* [*a* ↦ *b*])
unfolding *munion-def*
apply (*simp add*: *disjoint-dom*, *rule dagger-singleton-sep*)
using *assms* **by** *simp-all*

lemma *sep-singleton*: *y*>*0* ⟹ *sep* [*x* ↦ *y*]
unfolding *sep-def* **by** *auto*

lemma *dom-ar-Disjoint*:
assumes *Disjoint f*
shows *Disjoint* (*s* -⊲ *f*)
unfolding *Disjoint-def*
by (*metis Disjoint-def Locs-of-def assms f-in-dom-ar-subsume f-in-dom-ar-the-subsume*)

lemma *singleton-Disjoint*: *Disjoint* [*a* ↦ *b*]
by (*metis Disjoint-def dom-empty empty-iff l-inmapupd-dom-iff*)

lemma *disjoint-locs-locs-of-weaken*:
assumes *ab-f-disj*: *disjoint* (*locs-of a b*) (*locs f*)
and *yinf*: *y* ∈ *dom f*
and *nat1f*: *nat1-map f*
shows *disjoint* (*locs-of a b*) (*locs-of y* (*the* (*f y*)))
proof -
have ∗: (*locs-of y* (*the* (*f y*))) ⊆ *locs f*
unfolding *locs-def* **apply** (*simp add*: *nat1f*)
proof
fix *x* **assume** *x* ∈ *locs-of y* (*the* (*f y*))
then show *x* ∈ (⋃ *s*∈*dom f*. *locs-of s* (*the* (*f s*)))
using *yinf* **by** *auto*
qed
thus *?thesis* **by** (*metis Int-empty-right Int-left-commute*
ab-f-disj disjoint-def le-iff-inf)
qed

lemma *disjoint-comm*: *disjoint X Y* = *disjoint Y X*
unfolding *disjoint-def* **by** *auto*

lemma *unionm-singleton-Disjoint*:
assumes *anotinf*: *a* ∉ *dom f*
and *disjf*: *Disjoint f*
and *nat1f*: *nat1-map f*
and *nat1b*: *nat1 b*
and *disj*: *disjoint* (*locs-of a b*) (*locs f*)
shows *Disjoint* (*f* ∪*m* [*a* ↦ *b*])

```
unfolding Disjoint-def
proof (intro ballI impI)
 fix x y
 assume xindom: x ∈ dom (f ∪m [a ↦ b])
 and yindom: y ∈ dom (f ∪m [a ↦ b])
 and xnoty: x ≠ y

 have disjoint (locs-of x (the ((f ∪m [a ↦ b]) x))) (locs-of y (the ((f ∪m [a ↦ b]) y)))

 proof (cases x=a)
   assume xeqa: x = a
   then show ?thesis
   proof (cases y = a)
     assume yeqa: y = a
     then have False using xnoty xeqa by simp
     thus ?thesis ..
    next
    assume ynoteqa: y ≠ a

    have yinf: y ∈ dom f
       by (rule-tac g=[a ↦b] in unionm-in-dom-left, rule yindom, simp add: disj anotinf, simp add:
ynoteqa)
    from disj have disjoint (locs-of a b) (locs-of y (the (f y)))
    proof(rule disjoint-locs-locs-of-weaken)
      show y ∈ dom f by (rule yinf)
    next
      show nat1-map f by (rule nat1f)
    qed
    moreover have (locs-of x (the ((f ∪m [a ↦ b]) x))) = locs-of a b
      by (subst l-the-map-union-right, simp-all add: xeqa anotinf)
    moreover have (locs-of y (the ((f ∪m [a ↦ b]) y))) = (locs-of y (the (f y)))
         by (subst l-the-map-union-left, simp-all add: ynoteqa yinf anotinf)
    ultimately show ?thesis by simp
  qed
 next
  assume xnoteqa: x ≠ a
  then show ?thesis
  proof (cases y = a)
    assume yeqa: y = a
    have xinf: x ∈ dom f
      by (rule-tac g=[a ↦b] in unionm-in-dom-left, rule xindom,
        simp add: disj anotinf, simp add: xnoteqa)
    from disj have disjoint (locs-of x (the (f x))) (locs-of a b)
     proof(subst disjoint-comm, rule disjoint-locs-locs-of-weaken)
      show x ∈ dom f by (rule xinf)
    next
      show nat1-map f by (rule nat1f)
    qed
    moreover have (locs-of x (the ((f ∪m [a ↦ b]) x))) = (locs-of x (the (f x)))
       by (subst l-the-map-union-left, simp-all add: xnoteqa xinf anotinf)
    moreover have (locs-of y (the ((f ∪m [a ↦ b]) y))) = locs-of a b
      by (subst l-the-map-union-right, simp-all add: yeqa anotinf)
    ultimately show ?thesis by simp
   next
    assume ynoteqa: y ≠ a
    then show ?thesis
```

```
    proof -
      have xinf: x ∈ dom f
        by (rule-tac g=[a ↦b] in unionm-in-dom-left,
              rule xindom, simp add: disj anotinf, simp add: xnoteqa)
      have yinf: y ∈ dom f
        by (rule-tac g=[a ↦b] in unionm-in-dom-left,
              rule yindom, simp add: disj anotinf, simp add: ynoteqa)
      have disjoint (locs-of x (the (f x))) (locs-of y (the (f y)))
        by (metis Disjoint-def xinf yinf disjf l-locs-of-Locs-of-iff xnoty)
      moreover have  (locs-of x (the ((f ∪m [a ↦ b]) x))) =  (locs-of x (the (f x)))
        by (subst l-the-map-union-left, simp-all add: xnoteqa xinf anotinf)
      moreover have (locs-of y (the ((f ∪m [a ↦ b]) y))) = (locs-of y (the (f y)))
         by (subst l-the-map-union-left, simp-all add: ynoteqa yinf anotinf)
      ultimately show ?thesis by simp
    qed
  qed
 qed
thus disjoint (Locs-of (f ∪m [a ↦ b]) x) (Locs-of (f ∪m [a ↦ b]) y)
 unfolding Locs-of-def by (simp add: xindom yindom)
qed


lemma l-locs-of-dom-ar:
  assumes nat1f: nat1-map f
  and disj: Disjoint f
  and rinf: r ∈ dom f
  shows locs({r} -⊲ f) = locs f - locs-of r (the(f r))
proof -
   have nat1-ar: nat1-map ({r} -⊲ f) using nat1f by (rule dom-ar-nat1-map)
   have  (⋃ s∈dom ({r} -⊲ f). locs-of s (the (({r} -⊲ f) s))) =
         (⋃ s∈dom ({r} -⊲ f). locs-of s (the (f s)))
     by (simp add: f-in-dom-ar-the-subsume)
   also have ... =  (⋃ s∈(dom f - {r}). locs-of s (the (f s)))
    by (metis l-dom-dom-ar)
   also have ... =  (⋃ s∈(dom f). locs-of s (the (f s))) - locs-of r (the (f r))
   proof (rule UN-minus)
    show ∀ ·x∈dom f - {r}. locs-of x (the (f x)) ∩ locs-of r (the (f r)) = {}
    proof
      fix x assume xdom: x ∈ dom f - {r}
      then have xnotr: x ≠ r by blast
      have xinf: x∈ dom f using xdom by simp
      from disj show locs-of x (the (f x)) ∩ locs-of r (the (f r)) = {}
        unfolding Disjoint-def disjoint-def Locs-of-def
        by (auto simp: xdom xnotr xinf rinf)
    qed
   qed
   also have ... = locs f - locs-of r (the (f r)) by (simp add: locs-def nat1f)
   finally show ?thesis by (simp add: locs-def nat1f nat1-ar)
qed



lemma F1-inv-empty: F1-inv empty
unfolding F1-inv-def Disjoint-def sep-def nat1-map-def
```

by *auto*

lemma *dom-ar-F1-inv*:
assumes *inv*: *F1-inv f1*
shows *F1-inv* ({*l*} -⊲ *f1*)
proof -
from *inv* **show** *?thesis*
proof
assume *disjf1*: *Disjoint f1*
and *sepf1*: *sep f1*
and *nat1-mapf1*: *nat1-map f1*
and *finitef1*: *finite* (*dom f1*)
show *?thesis*
proof
show *nat1-conc*: *nat1-map* ({*l*} -⊲ *f1*) **using** *nat1-mapf1* **by** (*rule dom-ar-nat1-map*)
show *finite-conc*: *finite* (*dom* ({*l*} -⊲ *f1*)) **using** *finitef1* **by** (*rule dom-ar-finite*)
show *sep* ({*l*} -⊲ *f1*) **using** *sepf1* **by** (*rule dom-ar-sep*)
show *Disjoint* ({*l*} -⊲ *f1*) **using** *disjf1* **by** (*rule dom-ar-Disjoint*)
qed
qed
qed

lemma *dom-ar-locs*:
assumes *finite*(*dom f*)
and *nat1f*: *nat1-map f*
and *disj*: *Disjoint f*
and *lindom*: *l*∈*dom f*
shows *locs* ({*l*} -⊲ *f*) = (*locs f*) - *locs-of l* (*the* (*f l*))
proof -
have *locs* ({*l*} -⊲ *f*) = (⋃ *s*∈*dom* ({*l*} -⊲ *f*). *locs-of s* (*the* (({*l*} -⊲ *f*) *s*)))
proof -
have *nat1-map* ({*l*} -⊲ *f*) **using** *nat1f* **by** (*rule dom-ar-nat1-map*)
thus *?thesis* **unfolding** *locs-def* **by** *simp*
qed
also have ... = (⋃ *s*∈*dom* ({*l*} -⊲ *f*). *locs-of s* (*the* (*f s*)))
by (*simp add*: *f-in-dom-ar-the-subsume*)
also have ...= (⋃ *s*∈(*dom f* - {*l*}). *locs-of s* (*the* (*f s*)))
by (*simp add*: *l-dom-dom-ar*)
also have ... = (⋃ *s*∈*dom f*. *locs-of s* (*the* ((*f*) *s*))) - *locs-of l* (*the*(*f l*))
proof (*rule UN-minus*)
show ∀ ·*s*∈*dom f* - {*l*}. *locs-of s* (*the* (*f s*)) ∩ *locs-of l* (*the* (*f l*)) = {}
proof
fix *s* **assume** *sdom*: *s* ∈ *dom f* - {*l*}
then have *snotl*: *s* ≠ *l* **by** *blast*
have *sinf*: *s*∈ *dom f* **using** *sdom* **by** *simp*
from *disj* **show** *locs-of s* (*the* (*f s*)) ∩ *locs-of l* (*the* (*f l*)) = {}
unfolding *Disjoint-def disjoint-def Locs-of-def*
by (*auto simp*: *sdom snotl sinf lindom*)
qed
qed
finally show *?thesis* **by** (*simp add*: *locs-def nat1f*)
qed

lemma *nat1-map-upd*: *nat1-map f* $\Longrightarrow$ *nat1 y* $\Longrightarrow$ *nat1-map* ($f(x \mapsto y)$)
unfolding *nat1-def nat1-map-def* **by** *auto*

lemma *finite-map-upd*: *finite* (*dom f*) $\Longrightarrow$ *finite* (*dom* ($f(x \mapsto y)$))
by *auto*

lemma *min-or*: *min x y* = *x* $\vee$ *min x y* = *y* **by** (*metis min-def*)

lemma *sumsize2-mapupd*: *finite* (*dom f*) $\Longrightarrow x \notin$ *dom f* $\Longrightarrow$ *f* $\neq$ *empty* $\Longrightarrow$ *sum-size* ($f(x \mapsto y)$) = (*sum-size f*) + *y*
unfolding *sum-size-def* **apply** *simp*
by (*smt setsum-cong2*)

lemma *setsum-mapupd*: *finite* (*dom fa*) $\Longrightarrow$ *e* $\notin$ *dom fa* $\Longrightarrow$ *fa* $\neq$ *empty* $\Longrightarrow$($\sum x \in$*dom* (*fa*($e \mapsto r$)). *the* ((*fa*($e \mapsto r$)) *x*)) = ($\sum x \in$*dom fa*. *the* (*fa x*)) + *r*
apply *simp* **apply** (*subst add-commute*)
by (*smt setsum.F-cong*)

lemma *sumsize2-weakening*: *x* $\notin$ *dom f* $\Longrightarrow$ *finite* (*dom f*) $\Longrightarrow$ *y*>*0* $\Longrightarrow$ *sum-size* ($f(x \mapsto y)$) > *0*
unfolding *sum-size-def*
by *simp*

lemma *sum-size-singleton*: *sum-size* [$x \mapsto y$] = *y*
unfolding *sum-size-def*
by *simp*

lemma *setsum-dagger*: *dom f* $\cap$ *dom g* = {} $\Longrightarrow$*finite* (*dom f*) $\Longrightarrow$ ($\sum x \in$*dom f*. *the* ((*f* † *g*) *x*)) = ($\sum x \in$*dom f*. *the* (*f x*))
apply (*rule setsum-cong*)
apply *simp*
apply (*subst l-dagger-apply*)
by *auto*

lemma *sum-size-dagger-single*: *finite* (*dom f*) $\Longrightarrow$ *f* $\neq$ *empty* $\Longrightarrow$ *x* $\notin$ *dom f* $\Longrightarrow$*sum-size* (*f* † [$x \mapsto y$])
= (*sum-size f*) + *y*
unfolding *sum-size-def*
apply (*simp add*: *dagger-notemptyL*)
apply (*subst l-dagger-dom*)
apply (*subst setsum-Un-disjoint*)
apply (*simp*)
apply *simp*
apply *simp*
apply *simp*
apply (*subst setsum-dagger*)

apply *simp*
apply *simp*
by (*metis dagger-upd-dist map-upd-Some-unfold the.simps*)

lemma *sum-size-munion*: *finite* (*dom f*) ⟹ *finite* (*dom g*) ⟹ *f* ≠ *empty* ⟹ *g* ≠ *empty* ⟹ *dom f* ∩ *dom g* = {} ⟹*sum-size* (*f* ∪*m g*)
= (*sum-size f*) + (*sum-size g*)
unfolding *sum-size-def*
apply(*simp add*: *munion-notempty-left*)
apply (*unfold munion-def*)
apply *simp*
apply (*subst l-dagger-dom*)
apply (*subst setsum-Un-disjoint*)
apply (*simp*)
apply *simp*
apply *simp*
apply (*simp add*: *setsum-dagger*)
apply (*subst l-dagger-commute*)
apply *simp*
apply (*subst setsum-dagger*)
by *auto*

lemma *dagger-min*: *finite* (*dom f*) ⟹ *finite* (*dom g*) ⟹ *f* ≠ *empty* ⟹
g ≠ *empty* ⟹ *Min* (*dom* (*f* † *g*)) ∈ *dom f* ∨ *Min* (*dom* (*f* † *g*)) ∈ *dom g*
apply (*simp add*: *l-dagger-dom*)
apply (*subst Min-Un*)
apply *simp-all*
apply (*subst Min-Un*)
apply *simp-all*
by (*metis* (*mono-tags*) *Min-in domIff emptyE less-imp-le min-max.inf-absorb2 min-max.le-iff-inf not-le*)

lemma *min-loc-munion*: *finite* (*dom f*) ⟹ *finite* (*dom g*) ⟹ *f*≠*empty* ⟹
g ≠ *empty* ⟹ *dom f* ∩ *dom g* = {} ⟹ (*min-loc* (*f* ∪*m g*)) ∈ *dom f* ∨ (*min-loc* (*f* ∪*m g*)) ∈ *dom g*
proof -
assume *finf*: *finite* (*dom f*) **and** *fing*: *finite* (*dom g*) **and**
fnotemp: *f*≠*empty* **and** *gnotemp*: *g* ≠ *empty* **and** *disjoint-dom*: *dom f* ∩ *dom g* = {}
have *Min* (*dom* (*f* ∪*m g*)) ∈ *dom f* ∨ *Min* (*dom* (*f* ∪*m g*)) ∈ *dom g*
unfolding *munion-def*
apply (*simp add*: *disjoint-dom*)
apply (*rule dagger-min*)
by (*simp-all add*: *finf fing fnotemp gnotemp*)
then show *min-loc* (*f* ∪*m g*) ∈ *dom f* ∨ *min-loc* (*f* ∪*m g*) ∈ *dom g*
unfolding *min-loc-def*
by (*metis dagger-def dagger-notemp-munion disjoint-dom fnotemp map-add-None*)
qed

lemma *munion-min-loc-nonempty*: *dom f1* ∩ *dom f2* = {} ⟹ *finite* (*dom f1*) ⟹ *finite* (*dom f2*) ⟹ *f1* ≠ *empty* ⟹ *f2* ≠ *empty* ⟹ *min-loc* (*f1* ∪*m f2*) = *min* (*min-loc f1*) (*min-loc f2*)

unfolding *min-loc-def munion-def* **apply** (*simp add*: *dagger-notemptyL*)
by (*metis Min.union-idem l-dagger-dom dom-eq-empty-conv*)

lemma *munion-min-loc-emptyf2*: *f2 = empty* $\Longrightarrow$ *min-loc* (*f1* $\cup m$ *f2*) = *min-loc f1*
by (*metis Int-empty-right equals0D l-map-non-empty-dom-conv l-munion-apply*)

lemma *munion-min-loc-emptyf1*: *f1 = empty* $\Longrightarrow$ *min-loc* (*f1* $\cup m$ *f2*) = *min-loc f2*
by (*metis* (*full-types*) *domIff dom-eq-empty-conv inf-bot-left l-dagger-apply munion-def*)

lemma *dagger-min-loc-nonempty*: *dom f1* $\cap$ *dom f2* = {} $\Longrightarrow$ *finite* (*dom f1*) $\Longrightarrow$ *finite* (*dom f2*) $\Longrightarrow$ *f1* $\neq$ *empty* $\Longrightarrow$ *f2* $\neq$ *empty* $\Longrightarrow$ *min-loc* (*f1* † *f2*) = *min* (*min-loc f1*) (*min-loc f2*)
unfolding *min-loc-def* **apply** (*simp add*: *dagger-notemptyL*)
by (*metis Min.union-idem l-dagger-dom dom-eq-empty-conv*)

lemma *dagger-min-loc-emptyf2*: *f2 = empty* $\Longrightarrow$ *min-loc* (*f1* † *f2*) = *min-loc f1*
by (*metis dom-eq-empty-conv empty-iff l-dagger-apply*)

lemma *dagger-min-loc-emptyf1*: *f1 = empty* $\Longrightarrow$ *min-loc* (*f1* † *f2*) = *min-loc f2*
by (*metis* (*full-types*) *domIff l-dagger-apply*)

lemma *min-loc-singleton*: *min-loc* [$x \mapsto y$] = x
unfolding *min-loc-def*
by *simp*

lemma *min-loc-dagger*: *finite* (*dom f*) $\Longrightarrow$ *finite* (*dom g*) $\Longrightarrow$ *f* $\neq$ *empty* $\Longrightarrow$ *g* $\neq$ *empty* $\Longrightarrow$*min-loc* (*f* † *g*)
= *min* (*min-loc f*) (*min-loc g*)
unfolding *min-loc-def*
apply(*simp add*: *dagger-notemptyL*)
apply (*subst l-dagger-dom*)
apply (*subgoal-tac dom f* $\neq$ {})
apply (*subgoal-tac dom g* $\neq$ {})
apply (*rule Min-Un*)
apply (*simp-all*)
done

lemma *locs-unionm-singleton*:
assumes *nat1y*: *nat1 y*
and *nat1f*: *nat1-map f*
and *xnotf*: $x \notin$ *dom f*
shows *locs*(*f* $\cup m$ [$x \mapsto y$]) = *locs f* $\cup$ *locs-of x y*
proof -
have *locs*(*f* $\cup m$ [$x \mapsto y$]) = ($\bigcup s \in dom$ (*f* $\cup m$ [$x \mapsto y$]). *locs-of s* (*the* ((*f* $\cup m$ [$x \mapsto y$]) *s*)))
unfolding *locs-def*
apply (*subst unionm-singleton-nat1-map*)
apply (*simp-all del*: *nat1-def add*: *nat1y nat1f xnotf*)
done
also have ... = ($\bigcup s \in dom$ (*f*) $\cup$ {x}. *locs-of s* (*the* ((*f* $\cup m$ [$x \mapsto y$]) *s*)))
apply (*subst l-munion-dom*)
apply (*simp add*: *xnotf*)
by (*simp*)
also have ... = ($\bigcup s \in$ *insert x* (*dom* (*f*)). *locs-of s* (*the* ((*f* $\cup m$ [$x \mapsto y$]) *s*)))

```
    by simp
  also have ... = (locs-of x (the ((f ∪m [x ↦ y]) x)))
          ∪ (⋃ s∈dom (f). locs-of s (the ((f ∪m [x ↦ y]) s)))
  by (rule UN-insert)
  also have ... = (locs-of x y)
          ∪ (⋃ s∈dom (f). locs-of s (the ((f ∪m [x ↦ y]) s)))
    apply (subst l-munion-apply)
    apply (simp add: xnotf)
    by auto
  also have ... =  (locs-of x y)
          ∪ (⋃ s∈dom (f). locs-of s (the(f s)))
  proof -
    have (⋃ s∈dom f. locs-of s (the ((f ∪m [x ↦ y]) s))) =
        (⋃ s∈dom (f). locs-of s (the(f s)))
    proof (rule UN-cong, rule refl)
    fix s
    assume *: s ∈ dom f
    then show  locs-of s (the ((f ∪m [x ↦ y]) s)) = locs-of s (the (f s))
    proof (subst l-munion-apply, (simp add: xnotf))
     have s ∉ dom [x ↦ y] by (metis * dom-empty empty-iff l-inmapupd-dom-iff xnotf)
     then show locs-of s (the (if s ∈ dom [x ↦ y] then [x ↦ y] s else f s)) = locs-of s (the (f s))
      by simp
   qed
 qed
 thus ?thesis by simp
qed
 also have ... =  locs-of x y  ∪ locs f
 unfolding locs-def by (simp add: nat1f)
 finally show ?thesis  by auto
qed

lemma locs-of-minus:
b>0 ⟹ c > 0 ⟹ b<c ⟹ locs-of a b =  (locs-of a c) - (locs-of  (a+b) (c-b))
apply (simp add: b-locs-of-as-set-interval) by auto

end


theory HEAP1ProofsIJW
imports  HEAP1LemmasIJW
begin




lemma F1-inv-restr-unionm:
  assumes inv: F1-inv f and nat1s: nat1 s and l-in-dom: l ∈ dom f
  and f-bigger-s: the(f l) > s
  shows F1-inv (({l} -⊲ f) ∪m [l + s ↦ the(f l) - s])
  proof -
  from inv show ?thesis
  proof
  assume disjf1: Disjoint f
   and sepf1: sep f
```

```
 and nat1-mapf1: nat1-map f
 and finitef1: finite (dom f)

have disjoint-dom: l+s ∉ dom ({l} -◁ f)
proof (rule l-dom-ar-not-in-dom)
  show l+s ∉ dom f
  proof (rule l-plus-s-not-in-f)
    show F1-inv f and l ∈ dom f and s < the (f l) and nat1 s
    by (simp-all del: nat1-def add: inv nat1s l-in-dom f-bigger-s)
  qed
qed

have noteqls: ∀· x ∈ dom f. x + (the (f x)) ≠ l + s
proof
  fix x assume x-in-dom: x ∈ dom f
  show x + the (f x) ≠ l + s
  proof (cases x=l)
    assume x=l
    then show ?thesis using f-bigger-s by simp
   next
    assume xnotl: x ≠ l
    from disjf1 have locs-of x (the (f x)) ∩ locs-of l (the (f l)) = {}
     unfolding Disjoint-def disjoint-def Locs-of-def
      by (auto simp: x-in-dom xnotl l-in-dom)
    moreover have l+s - 1  ∈ locs-of l (the (f l))
      by (metis f-bigger-s nat1s top-locs-of2)
    moreover have x + the (f x) - 1 ∈ locs-of x (the (f x))
      by (metis nat1-map-def nat1-mapf1 top-locs-of x-in-dom)
    ultimately have x + the (f x) - 1 ≠ l+s - 1 by auto
    thus ?thesis by simp
  qed
qed
show ?thesis
proof
  from disjoint-dom show nat1-conc: nat1-map ({l} -◁ f ∪m [l + s ↦ the (f l) - s])
  proof(rule unionm-singleton-nat1-map)
    show nat1-map ({l} -◁ f) using nat1-mapf1 by (rule dom-ar-nat1-map)
   next
    show nat1 (the (f l) - s) using f-bigger-s by simp
  qed
 next
  from disjoint-dom show finite (dom ({l} -◁ f ∪m [l + s ↦ the (f l) - s]))
  proof (rule unionm-singleton-finite)
    show finite (dom ({l} -◁ f))
      using finitef1 by (rule dom-ar-finite)
  qed
 next


  from disjoint-dom show sep ({l} -◁ f ∪m [l + s ↦ the (f l) - s])
  proof (rule unionm-singleton-sep)
    show sep ({l} -◁ f) using sepf1 by (rule dom-ar-sep)
   next
    show ∀·la∈dom ({l} -◁ f). la + the (({l} -◁ f) la) ∉ dom [l + s ↦ the (f l) - s]
      by (metis dom-eq-singleton-conv f-in-dom-ar-subsume
          f-in-dom-ar-the-subsume noteqls singletonE)
```

```
    next
     show l + s + (the (f l) - s) ∉ dom ({l} -◁ f)
     proof -
       have myfact: l + the (f l)  ∉ dom(f) using l-in-dom sepf1 sep-def by auto
       have l + the (f l) ∉ dom({l} -◁ f) by (metis l-dom-ar-not-in-dom myfact)
       then show ?thesis by (smt f-bigger-s)
     qed
    next
     show nat1 (the (f l) - s)
       using nat1-mapf1 f-bigger-s by auto
   qed
  next

  show Disjoint ({l} -◁ f ∪m [l + s ↦ the (f l) - s])
  proof (rule unionm-singleton-Disjoint)
   show Disjoint ({l} -◁ f) using disjf1 by (rule dom-ar-Disjoint)
  next
   show nat1-map ({l} -◁ f)  using nat1-mapf1 by (rule dom-ar-nat1-map)
  next
   show nat1fminuss: nat1 (the (f l) - s) by (simp add: f-bigger-s)
  next
   show l + s ∉ dom ({l} -◁ f)
     by (metis f-bigger-s f-in-dom-ar-subsume inv l-in-dom l-plus-s-not-in-f nat1s)
  next


  have disjoint  (locs-of (l + s) (the (f l) - s))    (locs ({l} -◁ f))
   proof -
     have (locs-of (l + s) (the (f l) - s)) ⊆ locs-of l (the (f l))
      sorry
     moreover have locs-of l (the (f l)) ∩  (locs ({l} -◁ f)) = {}
      sorry
    ultimately show ?thesis
       by (smt Int-absorb1 Int-assoc Int-commute Int-empty-left disjoint-def)
  qed

   show disjoint (locs-of (l + s) (the (f l) - s)) (locs ({l} -◁ f))
   proof -
     have (locs-of (l + s) (the (f l) - s)) ⊆ locs-of l (the (f l))
       by (rule locs-of-subset,simp add: f-bigger-s)
     moreover have locs-of l (the (f l)) ∩  (locs ({l} -◁ f)) = {}
     proof (subst l-locs-of-dom-ar)
       show nat1-map f and Disjoint f and l ∈ dom f
         by (simp-all add: l-in-dom nat1-mapf1 disjf1)
       next
       show locs-of l (the (f l)) ∩ (locs f - locs-of l (the (f l))) = {}
           by simp
     qed
    ultimately show ?thesis
       by (smt Int-absorb1 Int-assoc Int-commute Int-empty-left disjoint-def)
   qed
  qed
 qed
qed
```

lemma (**in** *level1-new*) *new1-post-feaseq*:
assumes *pre-eq*: $\exists \cdot l \in dom\ f1.\ the\ (f1\ l) = s1$
shows $\exists \cdot\ r\ f1new.\ new1\text{-}post\text{-}eq\ f1\ s1\ f1new\ r \wedge F1\text{-}inv\ f1new$
proof -
from *pre-eq* **obtain** *l* **where** *ind*: $l \in dom\ f1$ **and** *preinstance*: $the\ (f1\ l) = s1$ **..**
obtain *f1new* **where** *f1wit*: $f1new = \{l\} \text{-}\lhd f1$ **by** *auto*
from *ind* **and** *preinstance* **and** *f1wit* **have** $l \in dom\ f1 \wedge the\ (f1\ l) = s1 \wedge f1new = \{l\} \text{-}\lhd f1$ **by** *simp*
moreover from *l1-invariant-def* **have** *F1-inv f1new* **by** (*simp only*: *dom-ar-F1-inv f1wit*)
ultimately show *?thesis* **using** *new1-post-eq-def* **by** *auto*
qed

lemma (**in** *level1-new*) *new1-post-feasgr*:
assumes *pre-gr*: $\exists \cdot l \in dom\ f1.\ the\ (f1\ l) > s1$
shows $\exists \cdot\ r\ f1new.\ new1\text{-}post\text{-}gr\ f1\ s1\ f1new\ r \wedge F1\text{-}inv\ f1new$
proof -
from *pre-gr* **obtain** *l* **where** *ind*: $l \in dom\ f1$ **and** *preinstance*: $the\ (f1\ l) > s1$ **..**
obtain *f1new* **where** *f1wit*: $f1new = (\{l\} \text{-}\lhd f1) \cup m\ [l + s1 \mapsto the(f1\ l) \text{-} s1]$ **by** *auto*
from *ind* **and** *preinstance* **and** *f1wit*
have $l \in dom\ f1 \wedge the\ (f1\ l) > s1 \wedge f1new = (\{l\} \text{-}\lhd f1) \cup m\ [l + s1 \mapsto the(f1\ l) \text{-} s1]$
by *simp*
moreover have *F1-inv f1new*
proof -
have *F1-inv* $((\{l\} \text{-}\lhd f1) \cup m\ [l + s1 \mapsto the(f1\ l) \text{-} s1])$
by (*rule F1-inv-restr-unionm*, *rule l1-invariant-def*, *rule l1-input-notempty-def*, *rule ind*, *rule pre-instance*)
then show *?thesis* **by** (*simp only*: *f1wit*)
qed
ultimately show *?thesis* **using** *new1-post-gr-def* **by** *auto*
qed

theorem (**in** *level1-new*)
locale1-new-FSB: *PO-new1-feasibility*

by (*metis le-neq-implies-less*
PO-new1-feasibility-def
new1-post-def new1-post-feaseq
new1-post-feasgr
new1-postcondition-def
new1-pre-defs
l1-new1-precondition-def)

```
lemma(in level1-dispose) disjoint-above-below[simp] :
      dom(dispose1-above f1 d1 s1) ∩ dom(dispose1-below f1 d1) = {}
unfolding dispose1-above-def dispose1-below-def
proof(rule l-dom-r-disjoint-weakening)
  show {x ∈ dom f1. x = d1 + s1} ∩ {x ∈ dom f1. x + the (f1 x) = d1} = {}
  proof (cases {x ∈ dom f1. x = d1 + s1} = {})
    assume {x ∈ dom f1. x = d1 + s1} = {}
    then show  {x ∈ dom f1. x = d1 + s1} ∩ {x ∈ dom f1. x + the (f1 x) = d1} = {}
      by auto
    next
    assume ∗: {x ∈ dom f1. x = d1 + s1} ≠ {}
    show  {x ∈ dom f1. x = d1 + s1} ∩ {x ∈ dom f1. x + the (f1 x) = d1} = {}
    proof (cases  {x ∈ dom f1. x + the (f1 x) = d1} = {})
    assume {x ∈ dom f1. x + the (f1 x) = d1} = {}
    then show   {x ∈ dom f1. x = d1 + s1} ∩ {x ∈ dom f1. x + the (f1 x) = d1} = {}
      by auto
    next
    assume ∗∗: {x ∈ dom f1. x + the (f1 x) = d1} ≠ {}
    show  {x ∈ dom f1. x = d1 + s1} ∩ {x ∈ dom f1. x + the (f1 x) = d1} = {}
    proof(rule ccontr)
      assume nonempty: {x ∈ dom f1. x = d1 + s1} ∩ {x ∈ dom f1. x + the (f1 x) = d1} ≠ {}
     from ∗ ∗∗ obtain x where xinter: x ∈ {x ∈ dom f1. x = d1 + s1} ∩ {x ∈ dom f1. x + the (f1
x) = d1}
         by (smt equals0I nonempty)
      from xinter have d1s1: x = d1 + s1 by auto
      from xinter have d1: x + the (f1 x) = d1 by auto
      from d1s1 d1 have d1 + s1 + the (f1 x) = d1 by auto
      then have s1 + the (f1 x) = 0 by auto
      then have False
         by (metis add-is-0 l1-input-notempty-def less-numeral-extra(3) nat1-def)
      thus False ..
    qed
  qed
 qed
qed

lemma (in level1-dispose) finite-dispose1-above: finite ( dom (dispose1-above f1 d1 s1))
unfolding dispose1-above-def
apply (rule l-dom-r-finite)
by (metis invF1-finite-weaken l1-invariant-def)

lemma (in level1-dispose) finite-dispose1-below: finite ( dom (dispose1-below f1 d1))
unfolding dispose1-below-def
apply (rule l-dom-r-finite)
by (metis invF1-finite-weaken l1-invariant-def)


lemma(in level1-dispose)  d1-not-dispose-above: d1 ∉ dom (dispose1-above f1 d1 s1)
unfolding dispose1-above-def
proof (subst l-dom-r-subseteq)
  show {x ∈ dom f1. x = d1 + s1} ⊆ dom f1
    by auto
  next
  show d1 ∉ {x ∈ dom f1. x = d1 + s1}
    by (smt l1-input-notempty-def mem-Collect-eq nat1-def)
qed
```

lemma (**in** *level1-dispose*) *d1-not-dispose-below*: *d1* ∉ *dom* (*dispose1-below f1 d1*)
unfolding *dispose1-below-def*
proof (*subst l-dom-r-subseteq*)
  show {*x* ∈ *dom f1*. *x* + *the* (*f1 x*) = *d1*} ⊆ *dom f1*
    by *auto*
  next
  show *d1* ∉ {*x* ∈ *dom f1*. *x* + *the* (*f1 x*) = *d1*}
    by (*metis* (*lifting*, *mono-tags*) *invF1-sep-weaken l1-invariant-def mem-Collect-eq sep-def*)
qed

lemma (**in** *level1-dispose*) *d1-not-above-below*: *d1* ∉ *dom* (*dispose1-above f1 d1 s1* ∪*m dispose1-below f1 d1*)
unfolding *munion-def*
  apply *simp*
  by (*metis* (*full-types*) *Un-iff d1-not-dispose-above d1-not-dispose-below l-dagger-dom*)

lemma (**in** *level1-dispose*) *dispose1-ext-union*: *dom* (*dispose1-ext f1 d1 s1*) =
    dom (*dispose1-above f1 d1 s1*) ∪ *dom* (*dispose1-below f1 d1*) ∪ {*d1*}
proof -
  have *dom* (*dispose1-ext f1 d1 s1*) = *dom* (*dispose1-above f1 d1 s1* ∪*m dispose1-below f1 d1*) ∪ *dom*([*d1* ↦ *s1*])
  unfolding *dispose1-ext-def*
  by (*rule l-munion-dom*, *simp add*: *d1-not-above-below*)
  also have ... = *dom*(*dispose1-above f1 d1 s1* † *dispose1-below f1 d1*) ∪ {*d1*}
    unfolding *munion-def* **by** *simp*
  finally show *?thesis* **by** (*simp add*: *l-dagger-dom*)
qed

lemma (**in** *level1-dispose*) *dispose1-ext-notempty*: *dispose1-ext f1 d1 s1* ≠ *Map.empty*
  by (*metis Un-commute Un-insert-left dispose1-ext-union dom-eq-empty-conv insert-not-empty*)

lemma (**in** *level1-dispose*) *dispose1-ext-dom-notempty*: *dom* (*dispose1-ext f1 d1 s1*) ≠ {}
by (*metis Un-insert-right dispose1-ext-union insert-not-empty*)

lemma (**in** *level1-dispose*) *d1notinf1*: *d1* ∉ *dom f1*
proof -
  have *dom f1* ⊆ *locs f1*
  proof(*rule domf-in-locs*)
    show *nat1-map f1* **by** (*metis invF1-nat1-map-weaken l1-invariant-def*)
  qed
  moreover have *d1* ∈ *locs-of d1 s1*
    unfolding *locs-of-def* **apply** (*simp only*: *l1-input-notempty-def*)
    by (*smt l1-input-notempty-def mem-Collect-eq nat1-def*)
  ultimately show *?thesis* **by** (*smt Collect-empty-eq Int-def disjoint-def*
            dispose1-pre-def l1-dispose1-precondition-def set-rev-mp)
qed

lemma (**in** *level1-dispose*) *min-loc-unfold*: *min-loc* (*dispose1-ext f1 d1 s1*)
  = *Min* ((*dom* (*dispose1-above f1 d1 s1*))
     ∪ (*dom* (*dispose1-below f1 d1*)) ∪ {*d1*})
proof -
  have *min-loc* (*dispose1-ext f1 d1 s1*) =
  min-loc (*dispose1-above f1 d1 s1* ∪*m dispose1-below f1 d1* ∪*m* [*d1* ↦ *s1*])

```
  unfolding dispose1-ext-def by simp
 also have ... =  Min (dom (dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1 ∪m [d1 ↦ s1]))
  unfolding min-loc-def
   by (fold dispose1-ext-def, simp add: dispose1-ext-notempty)
 also have  ...= Min ((dom (dispose1-above f1 d1 s1)) ∪ (dom (dispose1-below f1 d1)) ∪ {d1})
   by (fold dispose1-ext-def,simp add: dispose1-ext-union)
 finally show ?thesis by simp
qed

lemma above-dom:
 assumes above-notempty: (dispose1-above f1 d1 s1) ≠ empty
 shows dom (dispose1-above f1 d1 s1) = {d1 + s1}
 proof -
 have dispose1-above f1 d1 s1 = { x ∈ dom f1 . x = d1 + s1 } ◁ f1
   by (metis dispose1-above-def)
 then have  { x ∈ dom f1 . x = d1 + s1 } ≠ {}
   by (metis above-notempty l-dom-r-nothing)
 moreover have dom (dispose1-above f1 d1 s1) = {d1 + s1}
  unfolding dispose1-above-def
 proof (subst l-dom-r-iff)
   show {x ∈ dom f1. x = d1 + s1} ∩ dom f1 = {d1 + s1}
     by (metis Collect-conj-eq Collect-conv-if Collect-mem-eq
         calculation inf-commute singleton-conv)
 qed
 thus ?thesis .
qed

lemma above-min-loc:
assumes above-notempty: (dispose1-above f1 d1 s1) ≠ empty
shows min-loc (dispose1-above f1 d1 s1) = d1 + s1
   unfolding min-loc-def
   by (metis Min-singleton assms above-dom)

lemma above-d1s1-in-f1:
assumes above-notempty: (dispose1-above f1 d1 s1) ≠ empty
shows d1+s1 ∈ dom f1
proof -
  have dom (dispose1-above f1 d1 s1) ⊆ dom (f1)
  unfolding dispose1-above-def by (simp add: l-dom-r-dom-subseteq)
  moreover have {d1+s1} ⊆ dom f1 by (metis above-dom assms calculation)
  ultimately show ?thesis by auto
qed

lemma above-sumsize:
assumes above-notempty: (dispose1-above f1 d1 s1) ≠ empty
shows sum-size (dispose1-above f1 d1 s1) = the (f1 (d1 + s1))
unfolding sum-size-def
apply (simp add: above-notempty)
apply (subst above-dom)
apply (rule above-notempty)
unfolding dispose1-above-def
apply (subgoal-tac {x. x = d1 + s1 ∧ x ∈ dom f1} = {d1+s1})
apply (simp)
apply (subst f-in-dom-r-apply-elem)
apply simp-all
by (metis Collect-conj-eq Collect-mem-eq Int-empty-left
```

Int-insert-left-if1 above-d1s1-in-f1 assms singleton-conv)

lemma (**in** *level1-dispose*) *d1-above*:
$\forall \cdot x \in dom\ (dispose1\text{-}above\ f1\ d1\ s1).\ x > d1$
by (*metis* (*mono-tags*) *above-dom d1-not-dispose-above*
l-map-non-empty-has-elem-conv less-le not-add-less1 not-less singletonE)

lemma (**in** *level1-dispose*) *min-below-empty*:
assumes *below-empty*: $dom\ (dispose1\text{-}below\ f1\ d1) = \{\}$
shows $min\text{-}loc\ (dispose1\text{-}ext\ f1\ d1\ s1) = d1$
proof(*cases* $dom\ (dispose1\text{-}above\ f1\ d1\ s1) = \{\}$)
assume $dom\ (dispose1\text{-}above\ f1\ d1\ s1) = \{\}$
then show *?thesis*
by (*metis min-loc-unfold Min-singleton below-empty sup-bot-left*)
next
assume *above-notempty*: $dom\ (dispose1\text{-}above\ f1\ d1\ s1) \neq \{\}$
have $min\text{-}loc\ (dispose1\text{-}ext\ f1\ d1\ s1) = Min\ (dom\ (dispose1\text{-}above\ f1\ d1\ s1) \cup \{d1\})$
by (*simp add*: *below-empty min-loc-unfold*)
also have ... = $min\ (Min\ (dom\ (dispose1\text{-}above\ f1\ d1\ s1)))\ (Min(\{d1\}))$
by (*subst Min-Un*, *simp-all del*: *dom-eq-empty-conv add*: *finite-dispose1-above l-map-non-empty-dom-conv above-notempty*)
also have ... = $min\ \ (Min\ (dom\ (dispose1\text{-}above\ f1\ d1\ s1)))\ d1$ **by** *simp*
finally show *?thesis* **by** (*metis Min-singleton above-dom above-notempty*
l-map-non-empty-dom-conv le-add1 min-absorb1 min-max.inf-commute)
qed

lemma *dom-ar-disjoint*: $(dom\ f) \cap (Y) = \{\} \Longrightarrow (dom\ (X \mathrel{-\!\!\triangleleft} f)) \cap Y = \{\}$
by (*metis Diff-Int-distrib2 empty-Diff l-dom-dom-ar*)

lemma (**in** *level1-dispose*) *min-below-notempty*:
assumes *below-notempty*: $dom\ (dispose1\text{-}below\ f1\ d1) \neq \{\}$
shows $min\text{-}loc\ (dispose1\text{-}ext\ f1\ d1\ s1) \in dom\ (dispose1\text{-}below\ f1\ d1)$
proof -
have $Min\ (dom\ (dispose1\text{-}above\ f1\ d1\ s1) \cup dom\ (dispose1\text{-}below\ f1\ d1) \cup \{d1\})$
$= Min\ (dom\ (dispose1\text{-}below\ f1\ d1) \cup (dom\ (dispose1\text{-}above\ f1\ d1\ s1) \cup \{d1\}))$
by (*metis Un-insert-left min-loc-unfold sup-bot-left sup-commute*)
also have ... = $min\ \ (Min\ (dom\ (dispose1\text{-}below\ f1\ d1)))$
$(Min\ (dom\ (dispose1\text{-}above\ f1\ d1\ s1) \cup \{d1\}))$
by (*subst Min-Un*, *simp-all del*: *dom-eq-empty-conv*
add: *finite-dispose1-above finite-dispose1-below l-map-non-empty-dom-conv below-notempty*)
also have ... = $min\ (Min\ (dom\ (dispose1\text{-}below\ f1\ d1)))\ \ d1$
proof (*cases* $dom\ (dispose1\text{-}above\ f1\ d1\ s1) = \{\}$)
assume $dom\ (dispose1\text{-}above\ f1\ d1\ s1) = \{\}$ **thus** *?thesis* **by** *simp*
next
assume $dom\ (dispose1\text{-}above\ f1\ d1\ s1) \neq \{\}$ **then**
have $Min\ (dom\ (dispose1\text{-}above\ f1\ d1\ s1) \cup \{d1\}) = Min\ (\{d1{+}s1\} \cup \{d1\})$
by (*metis above-dom l-map-non-empty-dom-conv*)
thus *?thesis* **by** (*simp add*: *l1-input-notempty-def*)
qed
also have ... = $Min\ (dom\ (dispose1\text{-}below\ f1\ d1))$
proof -

```
 have ∗: ∃· x. x∈dom f1 ∧ x + the (f1 x) = d1
  proof -
    have dispose1-below f1 d1 ≠ empty by (metis below-notempty dom-eq-empty-conv)
    then have { x ∈ dom f1 . x + the(f1 x) = d1 } ≠ {}
     by (metis (full-types) dispose1-below-def l-dom-r-nothing)
    thus ?thesis  by (smt empty-Collect-eq)
  qed
 then obtain x where xinf1: x ∈ dom f1 and belowplusf1below: x + the (f1 x) = d1 by metis
 then have x < d1 by (metis antisym d1notinf1 leI le-add1)
 moreover have x ∈ dom (dispose1-below f1 d1)
 unfolding dispose1-below-def
 proof (subst l-dom-r-iff)
   show x ∈ {x ∈ dom f1. x + the (f1 x) = d1} ∩ dom f1
    by (smt Int-Collect  belowplusf1below xinf1 inf-commute)
 qed
 moreover have Min (dom (dispose1-below f1 d1)) < d1
   by (metis (full-types) Min-def all-not-in-conv
     calculation(1) calculation(2) finite-dispose1-below fold1-strict-below-iff)
 ultimately show ?thesis by (simp)
qed
also have ... ∈ dom (dispose1-below f1 d1)
 by (metis Min-in below-notempty finite-dispose1-below)
finally show ?thesis by (metis min-loc-unfold)
qed


lemma (in level1-dispose)
 nonzero-inter-dom:
   dom ((dom (dispose1-below f1 d1) ∪ dom (dispose1-above f1 d1 s1)) -◁ f1) ∩
     dom [min-loc (dispose1-ext f1 d1 s1) ↦ sum-size (dispose1-ext f1 d1 s1)]
   = {}
proof(cases dom (dispose1-below f1 d1) = {})
 assume below-empty: dom (dispose1-below f1 d1) = {}
 then have min-loc-shape: min-loc (dispose1-ext f1 d1 s1) = d1 by (rule min-below-empty)
 have dom-inter: dom f1 ∩ {d1} = {} by (metis Int-insert-left-if0 d1notinf1 inf-bot-left inf-commute)
 show ?thesis by (simp add: min-loc-shape dom-inter dom-ar-disjoint)
next
 assume below-notempty: dom (dispose1-below f1 d1) ≠ {}
 let ?S = (dom (dispose1-below f1 d1))
 let ?x = min-loc (dispose1-ext f1 d1 s1)
 have ?S ⊆ dom f1
   unfolding dispose1-below-def
   by (simp add: l-dom-r-dom-subseteq)
 moreover have ?x ∈ ?S by (metis below-notempty min-below-notempty)
 moreover have ?x ∉ dom (?S -◁ f1)
   by (metis calculation(2) l-dom-ar-notin-dom-or)
 moreover have ?x ∉ dom ((?S ∪ dom (dispose1-above f1 d1 s1)) -◁ f1)
   by (metis Un-iff calculation(2) l-dom-ar-not-in-dom2)
thus ?thesis by (metis Collect-conj-eq Collect-conv-if2 Int-commute dom-def
                dom-eq-singleton-conv mem-Collect-eq singleton-conv2)
qed

lemma (in level1-dispose) nat1-dispose1-ext:  nat1 (sum-size (dispose1-ext f1 d1 s1))
 unfolding dispose1-ext-def
         apply (subst l-munion-upd)
         apply (simp add: l-munion-dom)
```

apply (*rule conjI*)
apply (*rule d1-not-dispose-above*)
apply (*rule d1-not-dispose-below*)
apply (*unfold nat1-def*)
apply (*rule sumsize2-weakening*)
apply (*simp add*: *l-munion-dom*)
apply (*rule conjI*)
apply (*rule d1-not-dispose-above*)
apply (*rule d1-not-dispose-below*)
apply (*metis disjoint-above-below finite-Un finite-dispose1-above finite-dispose1-below l-munion-dom*)
by (*metis l1-input-notempty-def nat1-def*)

lemma (**in** *level1-dispose*) *F1-inv-dispose*:
assumes *f1inv*: *F1-inv f1*
shows *F1-inv* ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1* ∪*m*
[*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *sum-size* (*dispose1-ext f1 d1 s1*)])
proof -
from *f1inv* **show** *?thesis*
proof
assume *disjf1*: *Disjoint f1*
and *sepf1*: *sep f1*
and *nat1-mapf1*: *nat1-map f1*
and *finitef1*: *finite* (*dom f1*)
show *?thesis*
proof(*rule invF1-shape*)
from *nonzero-inter-dom* **show** *nat1-map* ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1*
∪*m* [*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *sum-size* (*dispose1-ext f1 d1 s1*)])
proof (*rule unionm-nat1-map*)
show *nat1-map* ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1*)
using *nat1-mapf1* **by** (*rule dom-ar-nat1-map*)
next
show *nat1-map* [*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1 s1*)]
by (*metis dom-empty empty-iff l-munion-empty-lhs nat1-dispose1-ext nat1-map-def unionm-singleton-nat1-map*)
qed
next
from *nonzero-inter-dom* **show** *finite* (*dom* ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1* ∪*m*
[*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *sum-size* (*dispose1-ext f1 d1 s1*)]))
proof (*rule unionm-finite*)
show *finite* (*dom* ((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1*))
by (*metis dom-ar-finite finitef1*)
next
show *finite* (*dom* [*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *sum-size* (*dispose1-ext f1 d1 s1*)])
by (*metis dom-empty dom-fun-upd finite.emptyI finite-insert option.distinct*(*1*))
qed
next
show *VDM-F1-inv*
((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1* ∪*m*
[*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *HEAP1.sum-size* (*dispose1-ext f1 d1 s1*)])
proof (*cases dispose1-below f1 d1 = empty*)
assume *below-empty*: *dispose1-below f1 d1 = empty*

```
  then show ?thesis
  proof (cases dispose1-above f1 d1 s1 = empty)
    assume above-empty: dispose1-above f1 d1 s1 = empty
    then show ?thesis
      unfolding dispose1-ext-def
   proof (simp add: below-empty l-munion-empty-rhs l-munion-empty-lhs l-dom-ar-empty-lhs min-loc-singleton
sum-size-singleton)
    show VDM-F1-inv (f1 ∪m [d1 ↦ s1])
    proof
      show sep (f1 ∪m [d1 ↦ s1])
      proof (rule unionm-singleton-sep)
        show sep f1 by (rule sepf1)
       next
        show ∀ ·l∈dom f1. l + the (f1 l) ∉ dom [d1 ↦ s1]
        proof
          fix l
          assume l ∈ dom f1
          have l + the (f1 l) ≠ d1
          proof -
            have dispose1-below f1 d1 = { x ∈ dom f1 . x + the(f1 x) = d1 } ◁ f1
             unfolding dispose1-below-def by simp
            then have { x ∈ dom f1 . x + the(f1 x) = d1 } = {}
              by (smt IntI below-empty dom-def dom-eq-empty-conv
                    empty-Collect-eq l-dom-r-iff mem-Collect-eq)
            thus ?thesis by (smt ‹l ∈ dom f1› empty-Collect-eq)
          qed
          then show l + the (f1 l) ∉ dom [d1 ↦ s1]
           by simp
        qed
        next
        show d1 + s1 ∉ dom f1
        proof -
          have dispose1-above f1 d1 s1 = { x ∈ dom f1 . x = d1 + s1 } ◁ f1
            unfolding dispose1-above-def by simp
          then have { x ∈ dom f1 . x = d1 + s1 } = {}
            by (smt Collect-empty-eq above-empty disjoint-iff-not-equal
                dom-def l-dom-r-iff mem-Collect-eq)
          thus ?thesis by (smt empty-Collect-eq)
        qed
        next
        show d1 ∉ dom f1 by (rule d1notinf1)
        next
        show nat1 s1 by (simp only: l1-input-notempty-def)
       qed
      next
       show Disjoint (f1 ∪m [d1 ↦ s1])
       proof (rule unionm-singleton-Disjoint)
         show Disjoint f1 by (rule disjf1)
        next
         show nat1-map f1 by (rule nat1-mapf1)
        next
         show d1 ∉ dom f1 by (rule d1notinf1)
        next
         show nat1 s1 by (rule l1-input-notempty-def)
        next
         from l1-dispose1-precondition-def show disjoint (locs-of d1 s1) (locs f1)
```

```
      unfolding dispose1-pre-def by assumption
    qed
  qed
qed
next
assume above-notempty: dispose1-above f1 d1 s1 ≠ Map.empty

have abovebelowshape: (dom (dispose1-below f1 d1) ∪ dom (dispose1-above f1 d1 s1)) = {d1+s1}
 by (simp add: below-empty l-munion-empty-rhs l-munion-empty-lhs l-dom-ar-empty-lhs d1notinf1
  d1-not-dispose-above d1-not-dispose-below above-dom above-notempty)
have min-loc-shape: min-loc (dispose1-ext f1 d1 s1) = d1
 by (metis below-empty l-map-non-empty-dom-conv min-below-empty)
have sum-size (dispose1-ext f1 d1 s1) = sum-size (dispose1-above f1 d1 s1) + s1
  by (simp add: dispose1-ext-def below-empty l-munion-empty-rhs
      l-munion-empty-lhs l-dom-ar-empty-lhs d1notinf1
      d1-not-dispose-above d1-not-dispose-below sum-size-munion
      finite-dispose1-above above-notempty sum-size-singleton)
then have sum-size-shape: sum-size (dispose1-ext f1 d1 s1) = the(f1 (d1+s1)) + s1
  by (simp add: above-sumsize above-notempty)
show ?thesis
proof (simp add: sum-size-shape min-loc-shape abovebelowshape)
  show VDM-F1-inv ({d1 + s1} -⊲ f1 ∪m [d1 ↦ the (f1 (d1 + s1)) + s1])
  proof
    show sep ({d1 + s1} -⊲ f1 ∪m [d1 ↦ the (f1 (d1 + s1)) + s1])
    proof (rule unionm-singleton-sep)
      show sep ({d1 + s1} -⊲ f1)  using sepf1 by (rule dom-ar-sep)
     next
      show  d1 ∉ dom ({d1 + s1} -⊲ f1)
      proof -
        have d1 ∉ dom f1 by (rule d1notinf1)
        thus ?thesis by (metis l-dom-ar-notin-dom-or)
      qed
     next
      show  ∀·l∈dom ({d1 + s1} -⊲ f1).
                l + the (({d1 + s1} -⊲ f1) l)
                  ∉ dom [d1 ↦ the (f1 (d1 + s1)) + s1]
      proof
        fix l assume lindom: l∈dom ({d1 + s1} -⊲ f1)
        then have linf: l∈ dom f1 by (metis l-dom-ar-notin-dom-or)
        have l + the (f1 l) ≠ d1
        proof -
          have dispose1-below f1 d1 = { x ∈ dom f1 . x + the(f1 x) = d1 } ⊲ f1
           unfolding dispose1-below-def by simp
          then have  { x ∈ dom f1 . x + the(f1 x) = d1 } = {}
            by (smt IntI below-empty dom-def dom-eq-empty-conv
                    empty-Collect-eq l-dom-r-iff mem-Collect-eq)
          thus ?thesis by (smt linf empty-Collect-eq)
        qed
        then have l + the (({d1 + s1} -⊲ f1) l) ≠ d1
          by (metis f-in-dom-ar-apply-subsume lindom)
        thus  l + the (({d1 + s1} -⊲ f1) l) ∉ dom [d1 ↦ the (f1 (d1 + s1)) + s1] by auto
      qed
     next
      show d1 + (the (f1 (d1 + s1)) + s1) ∉ dom ({d1 + s1} -⊲ f1)
      proof -
        have sep f1 by (rule sepf1)
```

```
        then have ∀·l∈ dom f1. l + the (f1 l) ∉ dom f1
          using sep-def by auto
        moreover have (d1+s1) ∈ dom f1 using above-notempty by (rule above-d1s1-in-f1)
        moreover have (d1 + s1) + the (f1 (d1+s1)) ∉ dom f1
           by (metis calculation(1) calculation(2))
        ultimately show d1 + (the (f1 (d1 + s1)) + s1) ∉ dom ({d1 + s1} -◁ f1)
          by (smt f-in-dom-ar-subsume)
       qed
      next
       show nat1 (the (f1 (d1 + s1)) + s1)
        by (simp, rule disjI2, metis l1-input-notempty-def nat1-def)
      qed
     next
     show Disjoint ({d1 + s1} -◁ f1 ∪m [d1 ↦ the (f1 (d1 + s1)) + s1])
      proof (rule unionm-singleton-Disjoint)
       show Disjoint ({d1 + s1} -◁ f1) using disjf1 by (rule dom-ar-Disjoint)
      next
       show d1 ∉ dom ({d1 + s1} -◁ f1) using d1notinf1 by (simp add: l-dom-ar-notin-dom-or)
      next
       show nat1-map ({d1 + s1} -◁ f1) using nat1-mapf1 by (rule dom-ar-nat1-map)
      next
       show nat1 (the (f1 (d1 + s1)) + s1)
         by (metis (mono-tags) add-eq-if l1-input-notempty-def nat1-def zero-less-Suc)
      next
       show disjoint (locs-of d1 (the (f1 (d1 + s1)) + s1)) (locs ({d1 + s1} -◁ f1))

       proof -
         from l1-dispose1-precondition-def have disjoint (locs-of d1 s1) (locs f1)
           by (simp add: dispose1-pre-def)
         moreover have (locs ({d1 + s1} -◁ f1)) = locs f1 - locs-of (d1+s1) (the (f1 (d1+s1)))
       by (rule dom-ar-locs, simp-all add: disjf1 finitef1 nat1-mapf1 above-d1s1-in-f1 above-notempty)
         moreover have locs-of d1 (the (f1 (d1 + s1)) + s1) =
                     locs-of d1 s1
                       ∪   locs-of (d1+s1) ((the (f1 (d1 + s1))))
         proof (subst add-commute, rule locs-of-sum-range)
           show nat1 (the (f1 (d1 + s1)))
            by (metis (full-types) above-d1s1-in-f1 above-notempty nat1-map-def nat1-mapf1)
          next
           show nat1 s1 by (metis l1-input-notempty-def)
         qed
         ultimately show ?thesis unfolding disjoint-def by auto
       qed
      qed
     qed
    qed
   qed
  qed
 next
  assume below-notempty: dispose1-below f1 d1 ≠ empty

  from below-notempty have ∃·x. x∈dom f1 ∧ x + the (f1 x) = d1
  proof -
    have dispose1-below f1 d1 ≠ empty by (rule below-notempty)
    then have { x ∈ dom f1 . x + the(f1 x) = d1 } ≠ {}
      by (metis (full-types) dispose1-below-def l-dom-r-nothing)
    thus ?thesis  by (smt empty-Collect-eq)
  qed
```

```
then obtain below where belowinf1: below∈dom f1
   and belowplusf1below: below + the (f1 below) = d1
   by metis
then have below-in-dom: below ∈ dom(dispose1-below f1 d1)
 unfolding dispose1-below-def
proof (subst l-dom-r-iff)
  show below ∈ {x ∈ dom f1. x + the (f1 x) = d1} ∩ dom f1
    by (smt Int-Collect belowinf1 belowplusf1below inf-commute)
qed
have below-shape: dispose1-below f1 d1 = [below ↦ the (f1 below)]
proof
  fix x
  show dispose1-below f1 d1 x = [below ↦ the (f1 below)] x
  proof (simp, intro allI impI conjI)
    from below-in-dom
    show dispose1-below f1 d1 below = Some (the (f1 below))
      unfolding dispose1-below-def
    proof (subst f-in-dom-r-apply-the-elem)
      show below ∈ dom f1 by (rule belowinf1)
     next
      show below ∈ {x ∈ dom f1. x + the (f1 x) = d1}
        by (smt belowinf1 belowplusf1below mem-Collect-eq)
    qed(rule refl)
   next
    assume xnoteqbelow: x ≠ below
    show dispose1-below f1 d1 x = None
    proof(rule ccontr)
      assume dispose1-below f1 d1 x ≠ None then
      have con: x ∈ dom (dispose1-below f1 d1)
        by auto
      from con have xindomrset: x ∈ {x ∈ dom f1. x + the (f1 x) = d1}
       unfolding dispose1-below-def
        by (metis (full-types) l-in-dom-dom-r)
      then have xinf: x∈ dom f1 by (simp add: xindomrset)
      have xeqd1: x + the (f1 x) = d1
       by (metis (lifting, mono-tags) mem-Collect-eq xindomrset)

      from disjf1 have ∗: locs-of x (the (f1 x)) ∩  locs-of below (the (f1 below)) = {}
        by (metis xnoteqbelow belowinf1 Disjoint-def
            disjoint-def l-locs-of-Locs-of-iff xinf)
      have nat1below:  nat1 (the (f1 below)) by (metis nat1-map-def nat1-mapf1 belowinf1)
      have nat1x:  nat1 (the (f1 x)) by (metis nat1-map-def nat1-mapf1 xinf)
      from xinf xeqd1 belowplusf1below belowinf1 nat1x nat1below
      have ∗∗: locs-of x (the (f1 x)) ∩  locs-of below (the (f1 below)) ≠ {}
        by (metis IntI ex-in-conv top-locs-of)
      from ∗ ∗∗ show False by simp
    qed
  qed
qed
then have dom-below: dom (dispose1-below f1 d1) = {below} by simp
have sum-size-below: sum-size (dispose1-below f1 d1) = the (f1 below)
 by (simp add: sum-size-singleton below-shape)

show ?thesis
proof (cases dispose1-above f1 d1 s1 = empty)
  assume above-empty:  dispose1-above f1 d1 s1 = empty
```

```
have abovebelow-shape: (dom (dispose1-below f1 d1)) ∪ (dom (dispose1-above f1 d1 s1))
                       = {below}
  by (simp add: above-empty dom-below)
have min-loc-shape: min-loc (dispose1-ext f1 d1 s1) = below
  by (metis dom-below insert-not-empty min-below-notempty singleton-iff)
have sum-size-shape: sum-size (dispose1-ext f1 d1 s1) = the (f1 below) + s1
 unfolding dispose1-ext-def
  by (simp add: above-empty l-munion-empty-lhs sum-size-munion sum-size-singleton
          finite-dispose1-below below-notempty d1-not-dispose-below sum-size-below)
show ?thesis
proof (simp add: sum-size-shape min-loc-shape abovebelow-shape)
show VDM-F1-inv ({below} -⊲ f1 ∪m [below ↦ the (f1 below) + s1])
proof
show sep ({below} -⊲ f1 ∪m [below ↦ the (f1 below) + s1])
  proof (rule unionm-singleton-sep)
    show sep ({below} -⊲ f1) using sepf1 by (rule dom-ar-sep)
   next
    show  below ∉ dom ({below} -⊲ f1) by (metis f-in-dom-ar-notelem)
   next
    show ∀ ·l∈dom ({below} -⊲ f1).
       l + the (({below} -⊲ f1) l) ∉ dom [below ↦ the (f1 below) + s1]
    proof
      fix l assume lin-restr-dom:l ∈ dom ({below} -⊲ f1)
      have l ∈ dom f1 using lin-restr-dom by (metis l-dom-ar-not-in-dom)
      have l + the (({below} -⊲ f1) l) ≠ below
        by (metis ‹l ∈ dom f1› belowinf1 f-in-dom-ar-apply-subsume
          lin-restr-dom sep-def sepf1)
      thus l + the (({below} -⊲ f1) l) ∉ dom [below ↦ the (f1 below) + s1]
        by (metis dom-empty empty-iff l-inmapupd-dom-iff)
    qed
   next
    show  below + (the (f1 below) + s1) ∉ dom ({below} -⊲ f1)
    proof -
      have below + (the (f1 below)) = d1
       by (metis belowplusf1below)
      then have d1 +s1 ∉ dom ({below} -⊲ f1)
      proof -
        have dispose1-above f1 d1 s1 =  { x ∈ dom f1 . x = d1 + s1 } ⊲ f1
          unfolding dispose1-above-def by simp
        then have  { x ∈ dom f1 . x = d1 + s1 } = {}
          by (smt Collect-empty-eq above-empty disjoint-iff-not-equal
            dom-def l-dom-r-iff mem-Collect-eq)
        thus ?thesis by (metis Collect-conj-eq Collect-mem-eq
             Un-empty-left f-in-dom-ar-apply-not-elem l-dom-ar-nothing
             domIff l-dom-ar-not-in-dom2 f-in-dom-ar-notelem inf-commute
             insert-def sup-commute)
      qed
      thus ?thesis by (metis belowplusf1below nat-add-commute nat-add-left-commute)
    qed
   next
    show nat1 (the (f1 below) + s1) by (metis nat1-dispose1-ext sum-size-shape)
  qed
 next
  show Disjoint ({below} -⊲ f1 ∪m [below ↦ the (f1 below) + s1])
  proof(rule unionm-singleton-Disjoint)
    show below ∉ dom ({below} -⊲ f1)
```

```
        by (simp add: below-in-dom dom-below l-dom-ar-notin-dom-or)
       next
        show Disjoint ({below} -◁ f1) using disjf1 by (rule dom-ar-Disjoint)
       next
        show nat1-map ({below} -◁ f1) using nat1-mapf1 by (rule dom-ar-nat1-map)
       next
        show  nat1 (the (f1 below) + s1) by (metis nat1-dispose1-ext sum-size-shape)
       next

        show disjoint (locs-of below (the (f1 below) + s1)) (locs ({below} -◁ f1))
        proof -
        from l1-dispose1-precondition-def have disjoint (locs-of d1 s1) (locs f1)
          by (simp add: dispose1-pre-def)
        moreover have (locs ({below} -◁ f1)) = locs f1 - locs-of below (the (f1 (below)))
         by (rule dom-ar-locs, simp-all add: disjf1 finitef1 nat1-mapf1 belowinf1)
        moreover have locs-of below (the (f1 below) + s1) =
                      locs-of below  (the (f1 below))
                      ∪ locs-of (below +  (the (f1 below))) s1
        proof (rule locs-of-sum-range)
          show nat1 (the (f1 below)) by (metis belowinf1 nat1-map-def nat1-mapf1)
         next
          show nat1 s1 by (rule l1-input-notempty-def)
        qed
        moreover have  locs-of (below +  (the (f1 below))) s1 = locs-of d1 s1
         by (metis belowplusf1below)
        ultimately show ?thesis unfolding disjoint-def by auto
      qed
     qed
    qed
   qed
   next
    assume above-notempty:  dispose1-above f1 d1 s1 ≠ Map.empty
    have above-below-shape: (dom (dispose1-below f1 d1) ∪ dom (dispose1-above f1 d1 s1))
                          = {below,d1+s1}
     by (metis Un-insert-left above-dom above-notempty dom-below sup-bot-left)
    have min-loc-shape:  min-loc (dispose1-ext f1 d1 s1)  = below
      by (metis dom-below insert-not-empty min-below-notempty singleton-iff)
    have sum-size-shape: sum-size  (dispose1-ext f1 d1 s1)
                                 =   the (f1 (d1 + s1)) + the (f1 below) + s1
    proof -
      have sum-size-above-below: sum-size (dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1)
                               =  the (f1 (d1 + s1)) + the (f1 below)
       by (simp add: sum-size-munion  finite-dispose1-above finite-dispose1-below above-notempty
          below-notempty above-sumsize sum-size-below)
      then show ?thesis unfolding dispose1-ext-def
      proof (subst sum-size-munion)
        show finite (dom (dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1)) and
             finite (dom [d1 ↦ s1])
         by (simp-all add:  finite-dispose1-above finite-dispose1-below k-finite-munion)
       next
        show  dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1 ≠ empty
         and [d1 ↦ s1] ≠ empty
          by (auto simp: munion-notempty-right below-notempty)
       next
        from d1-not-above-below show dom (dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1) ∩ dom
[d1 ↦ s1] = {}
```

```
      by simp
    next
     show sum-size (dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1) + sum-size [d1 ↦ s1]
          = the (f1 (d1 + s1)) + the (f1 below) + s1
       by (simp add: sum-size-above-below sum-size-singleton)
   qed
  qed
  show ?thesis
  proof (simp add: sum-size-shape min-loc-shape above-below-shape)
   show VDM-F1-inv ({below, d1 + s1} -⊲ f1 ∪m [below ↦ the (f1 (d1 + s1)) + the (f1 below) +
s1])
   proof
    show sep ({below, d1 + s1} -⊲ f1 ∪m [below ↦ the (f1 (d1 + s1)) + the (f1 below) + s1])
    proof (rule unionm-singleton-sep)
     show sep ({below, d1 + s1} -⊲ f1) using sepf1 by (rule dom-ar-sep)
    next
     show below ∉ dom ({below, d1 + s1} -⊲ f1)
      by (metis insertI1 l-dom-ar-notin-dom-or)
    next
     show ∀·l∈dom ({below, d1 + s1} -⊲ f1).
          l + the (({below, d1 + s1} -⊲ f1) l)
          ∉ dom [below ↦ the (f1 (d1 + s1)) + the (f1 below) + s1]
      proof
       fix l assume lin-restr-dom: l ∈ dom ({below, d1 + s1} -⊲ f1)
       have l ∈ dom f1 using lin-restr-dom by (metis l-dom-ar-not-in-dom)
       have l + the (({below, d1 + s1} -⊲ f1) l) ≠ below
        by (metis ‹l ∈ dom f1› belowinf1 f-in-dom-ar-apply-subsume
           lin-restr-dom sep-def sepf1)
      thus l + the (({below,d1+s1} -⊲ f1) l) ∉ dom [below ↦ the (f1 (d1 + s1)) + the (f1 below)
+ s1]
        by (metis dom-empty empty-iff l-inmapupd-dom-iff)
     qed
    next
     show below + (the (f1 (d1 + s1)) + the (f1 below) + s1) ∉ dom ({below, d1 + s1} -⊲ f1)
     proof -
       have below + (the (f1 below)) = d1
        by (metis belowplusf1below)
       then have (d1 +s1) + (the (f1 (d1 + s1))) ∉ dom ({below,d1+s1} -⊲ f1)
         by (metis above-d1s1-in-f1 above-notempty l-dom-ar-notin-dom-or sep-def sepf1)
       thus ?thesis by (smt belowplusf1below)
     qed
    next
     show nat1 (the (f1 (d1 + s1)) + the (f1 below) + s1)
      by (metis nat1-dispose1-ext sum-size-shape)
    qed
   next
    show Disjoint ({below, d1 + s1} -⊲ f1 ∪m [below ↦ the (f1 (d1 + s1)) + the (f1 below) + s1])
     proof (rule unionm-singleton-Disjoint)
       show below ∉ dom ({below, d1 + s1} -⊲ f1)
        by (metis insertI1 l-dom-ar-notin-dom-or)
      next
       show Disjoint ({below, d1 + s1} -⊲ f1) using disjf1 by (rule dom-ar-Disjoint)
      next
       show nat1-map ({below, d1 + s1} -⊲ f1) using nat1-mapf1 by (rule dom-ar-nat1-map)
      next
       show nat1 (the (f1 (d1 + s1)) + the (f1 below) + s1)
```

```
  by (metis nat1-dispose1-ext sum-size-shape)
 next

 show
 disjoint (locs-of below  (the (f1 (d1 + s1)) +  the (f1 below) + s1))
       (locs ( {below, d1 + s1} -◁ f1))
 proof -
  have (locs ({below, d1 + s1} -◁ f1)) = locs ( {below} -◁ ({d1 + s1} -◁ f1))
   by (metis Un-empty-left Un-insert-left l-dom-ar-accum)
  also have ... =
     locs ({d1 + s1} -◁ f1) - locs-of below (the (({d1 + s1} -◁ f1) below))
  proof (rule dom-ar-locs)
   show finite (dom ({d1 + s1} -◁ f1)) by (metis dom-ar-finite finitef1)
  next
   show nat1-map ({d1 + s1} -◁ f1) by (metis dom-ar-nat1-map nat1-mapf1)
  next
   show Disjoint ({d1 + s1} -◁ f1) by (metis disjf1 dom-ar-Disjoint)
  next
   show  below ∈ dom ({d1 + s1} -◁ f1) by (metis belowinf1 belowplusf1below d1notinf1
                                           inf-commute inf-min l-in-dom-ar nat-min-absorb1
                                           singletonE)
  qed
  also have ... = locs ({d1 + s1} -◁ f1) - locs-of below (the (f1 below))
  proof (subst f-in-dom-ar-apply-not-elem)
   show below ∉ {d1 + s1}
    by (metis belowinf1 belowplusf1below d1notinf1 empty-iff insert-iff nat-neq-iff not-add-less1)

  qed (rule refl)
 also have ... = locs(f1) - locs-of (d1+s1) (the (f1 (d1+s1))) - locs-of below (the (f1 below))
    by(subst dom-ar-locs, simp-all add: disjf1 finitef1 nat1-mapf1 belowinf1 above-d1s1-in-f1
above-notempty )
 finally have *: (locs ({below, d1 + s1} -◁ f1))
          =  locs(f1) - locs-of (d1+s1) (the (f1 (d1+s1)))
           - locs-of below (the (f1 below)) by simp
 have locs-of below (the (f1 (d1 + s1)) + the (f1 below) + s1) =
      locs-of below (the (f1 below) + ((the (f1 (d1 + s1))) + s1))
  by (metis nat-add-commute nat-add-left-commute)
 also have ...  = (locs-of below (the (f1 below))) ∪
          (locs-of (below + (the (f1 below))) (the (f1 (d1 + s1)) + s1))
 proof (rule locs-of-sum-range)
  show nat1 (the (f1 below)) by (metis belowinf1 nat1-map-def nat1-mapf1)
 next
  show nat1 (the (f1 (d1 + s1)) + s1)
     by (metis (full-types) l1-input-notempty-def nat1-def trans-less-add2)
 qed
 also have ... = (locs-of below (the (f1 below))) ∪
              (locs-of d1 (the (f1 (d1 + s1)) + s1))
  by (metis belowplusf1below)
 also have ... =  (locs-of below (the (f1 below))) ∪
              locs-of d1 (s1+ (the (f1 (d1 + s1))))
  by (metis nat-add-commute)
 also have ... =  (locs-of below (the (f1 below)))
              ∪ locs-of d1 s1
              ∪ locs-of (d1+s1) (the (f1 (d1 + s1)))
 proof (subst locs-of-sum-range)
  show nat1 s1 by (metis l1-input-notempty-def)
```

```
        next
         show nat1 (the (f1 (d1 + s1)))
           by (metis above-d1s1-in-f1 above-notempty nat1-map-def nat1-mapf1)
       qed (metis sup-commute sup-left-commute)
       finally have **: locs-of below (the (f1 (d1 + s1)) + the (f1 below) + s1)
                    =(locs-of below (the (f1 below)))
                      ∪ locs-of d1 s1
                      ∪ locs-of (d1+s1) (the (f1 (d1 + s1)))
         by simp
       from l1-dispose1-precondition-def have ***: disjoint (locs-of d1 s1) (locs f1)
           by (simp add: dispose1-pre-def)
       from * ** *** show ?thesis unfolding disjoint-def by auto

     qed
   qed
  qed
 qed
qed
qed
qed
qed
 qed


theorem (in level1-dispose)
  locale1-dispose-FSB: PO-dispose1-feasibility
unfolding PO-dispose1-feasibility-def dispose1-postcondition-def
proof(subst dispose1-equiv)
 obtain f1new where f1wit: f1new = (dom (dispose1-below f1 d1) ∪ dom (dispose1-above f1 d1 s1))
-◁ f1 ∪m
            [min-loc (dispose1-ext f1 d1 s1) ↦ sum-size (dispose1-ext f1 d1 s1)]
 by auto
  from f1wit F1-inv-dispose show ∃·f'. dispose1-post2 f1 d1 s1 f' ∧ F1-inv f'
  using dispose1-post2-def by (metis l1-invariant-def)
qed



end

theory HEAP1SanityIJW
imports HEAP1ProofsIJW
begin

lemma new1-dispose-1-identity-isar:
  assumes nat1n: nat1 n
  and n1-post: new1-post f n f' r
  and d1-post: dispose1-post f' r n f''
  and inv: F1-inv f
  shows f = f''
proof -
  from n1-post show ?thesis
  unfolding new1-post-def
  proof
    assume n1-eq: new1-post-eq f n f' r
    then show f = f''
```

```
  unfolding new1-post-eq-def
proof(elim conjE)
  assume r-in-dom: r ∈ dom f
  and eq-n: the (f r) = n
  and f'-restr: f' = {r} -◁ f

  have below-shape: dispose1-below ({r} -◁ f) r = empty
    unfolding dispose1-below-def
    proof (rule l-dom-r-nothing-empty)
      from inv show {x ∈ dom ({r} -◁ f). x + the (({r} -◁ f) x) = r} = {}
      proof
        assume sepf: sep f
          have {x ∈ dom ({r} -◁ f). x + the (({r} -◁ f) x) = r}
             = {x ∈ dom ({r} -◁ f). x + the (f x) = r}
              by (metis (full-types) Diff-iff l-dom-dom-ar f-in-dom-ar-apply-not-elem)
          also have ... ⊆ {x ∈ dom (f). x + the (f x) = r}
            by (smt Collect-empty-eq l-dom-ar-not-in-dom r-in-dom sep-def sepf subsetI)
          also have ... = {}
            by (smt Collect-empty-eq r-in-dom sep-def sepf)
        finally show ?thesis by simp
      qed
    qed

  have above-shape: dispose1-above ({r} -◁ f) r n = empty
  unfolding dispose1-above-def

      proof (rule l-dom-r-nothing-empty)
      from inv show {x ∈ dom ({r} -◁ f). x = r + n} = {}
      proof
        assume sepf: sep f
          have {x ∈ dom ({r} -◁ f). x = r + n} ⊆ {x ∈ dom (f). x = r + n}
            by (smt Collect-cong Collect-empty-eq equals0D l-dom-ar-not-in-dom subsetI)
          also have ... = {} by (smt empty-Collect-eq eq-n r-in-dom sep-def sepf)
        finally show ?thesis by simp
      qed
    qed

  have min-loc-shape:  min-loc (dispose1-ext ({r} -◁ f) r n) = r
    unfolding dispose1-ext-def
  proof -
    have dispose1-above ({r} -◁ f) r n ∪m dispose1-below ({r} -◁ f) r ∪m [r ↦ n]
        = [r ↦ n]
         by (simp add: above-shape l-munion-empty-lhs below-shape)
    moreover have min-loc [r ↦ n] = r
      unfolding min-loc-def
      by simp
      ultimately show min-loc (dispose1-above ({r} -◁ f) r n
                          ∪m dispose1-below ({r} -◁ f) r ∪m [r ↦ n]) = r
       by simp
  qed

  have sum-size-shape:  sum-size (dispose1-ext ({r} -◁ f) r n) = the(f r)
    unfolding dispose1-ext-def
  proof -
    have dispose1-above ({r} -◁ f) r n ∪m dispose1-below ({r} -◁ f) r ∪m [r ↦ n]
        = [r ↦ n]
```

```
          by (simp add: above-shape l-munion-empty-lhs below-shape)
      moreover have sum-size [r ↦ n] = n
        unfolding sum-size-def by simp
      moreover have the (f r) = nby (rule eq-n)
      ultimately show sum-size (dispose1-above ({r} -◁ f) r n
            ∪m dispose1-below ({r} -◁ f) r ∪m [r ↦ n]) = the (f r)
          by simp
    qed

    from d1-post show ?thesis
    proof (simp only: dispose1-equiv, unfold dispose1-post2-def)
      assume f'' = (dom (dispose1-below f' r) ∪ dom (dispose1-above f' r n)) -◁ f' ∪m
        [min-loc (dispose1-ext f' r n) ↦ sum-size (dispose1-ext f' r n)]
      then have f'' = ((dom (empty) ∪ dom (empty)) -◁ f'
                ∪m [min-loc (dispose1-ext f' r n) ↦ sum-size (dispose1-ext f' r n)])
          by (simp add: f'-restr below-shape above-shape)
      then have f'' = {} -◁ f'
                ∪m [min-loc (dispose1-ext f' r n) ↦ sum-size (dispose1-ext f' r n)]
                by simp
      also have  ... =  f'
                ∪m [min-loc (dispose1-ext f' r n) ↦ sum-size (dispose1-ext f' r n)]
          by (metis l-dom-ar-empty-lhs)
      also have ...= f' ∪m [r ↦ the (f r)]
        by (simp add: min-loc-shape sum-size-shape f'-restr)
      also have ... = ({r} -◁ f) ∪m [r ↦ the (f r)]
        by (simp add: f'-restr)
      also have ... =  ({r} -◁ f) † [r ↦ the (f r)]
        proof -
          have dom ({r} -◁ f) ∩ dom [r ↦ the (f r)] = {}
            by (metis Int-insert-right-if0 dom-eq-singleton-conv f-in-dom-ar-notelem inf-bot-right)
          thus ?thesis by (simp add: munion-def)
        qed
      also have ... = f using r-in-dom by (rule antirestr-then-dagger)
      finally show ?thesis ..
    qed
  qed
  next
    assume new1-post-gr f n f' r
    then show ?thesis
      unfolding new1-post-gr-def
    proof(elim conjE)
      assume r-in-dom: r ∈ dom f
      and gr-n: the (f r) > n
      and f'-restr: f' = {r} -◁ f ∪m [r + n ↦ the (f r) - n]

      have disjoint-dom:  dom f ∩ dom [r + n ↦ the (f r) - n] = {}
      proof (simp)
      show r + n ∉ dom  f
      proof (rule l-plus-s-not-in-f)
        show F1-inv f by (metis inv)
        next
        show r ∈ dom  f by (rule r-in-dom)
        next
        show n < the (f r) by (rule gr-n)
        next
        show nat1 n by (rule nat1n)
```

```
    qed
  qed
    have disjoint-dom-antirestr:  dom ({r} -◁ f) ∩ dom [r + n ↦ the (f r) - n] = {}
     by (metis disjoint-dom l-dom-ar-disjoint-weakening)

    have below-shape: dispose1-below ({r} -◁ f ∪m [r + n ↦ the (f r) - n]) r = empty
      unfolding dispose1-below-def
     proof (rule l-dom-r-nothing-empty)
      from inv show {x ∈ dom ({r} -◁ f ∪m [r + n ↦ the (f r) - n]).
            x + the (({r} -◁ f ∪m [r + n ↦ the (f r) - n]) x) = r} = {}
      proof
       assume sepf: sep f
        have ({r} -◁ f ∪m [r + n ↦ the (f r) - n])
                    =({r} -◁ f † [r + n ↦ the (f r) - n])
                  by (metis  disjoint-dom-antirestr munion-def)
        then have {x ∈ dom ({r} -◁ f ∪m [r + n ↦ the (f r) - n]).
           x + the (({r} -◁ f ∪m [r + n ↦ the (f r) - n]) x) = r}
          = {x ∈ dom ({r} -◁ f † [r + n ↦ the (f r) - n]).
          x + the (({r} -◁ f † [r + n ↦ the (f r) - n]) x) = r}
           by simp
       also have ... = {x ∈ dom ({r} -◁ f).
          x + the (({r} -◁ f) x) = r} ∪ {x∈ dom ([r + n ↦ the (f r) - n]).
                x + the  ([r + n ↦ the (f r) - n] x) = r}
       proof (subst l-dagger-dom)
         show {x ∈ dom ({r} -◁ f) ∪ dom [r + n ↦ the (f r) - n].
                x + the (({r} -◁ f † [r + n ↦ the (f r) - n]) x) = r}
           =   {x ∈ dom ({r} -◁ f). x + the (({r} -◁ f) x) = r}
            ∪ {x ∈ dom [r + n ↦ the (f r) - n]. x + the ([r + n ↦ the (f r) - n] x) = r}
         proof (subst union-comp)
          show {x ∈ dom ({r} -◁ f). x + the (({r} -◁ f † [r + n ↦ the (f r) - n]) x) = r}
             ∪ {x ∈ dom [r + n ↦ the (f r) - n]. x + the (({r} -◁ f † [r + n ↦ the (f r) - n])
x) = r}
           =   {x ∈ dom ({r} -◁ f). x + the (({r} -◁ f) x) = r}
            ∪ {x ∈ dom [r + n ↦ the (f r) - n]. x + the ([r + n ↦ the (f r) - n] x) = r}
          proof -
           have  {x ∈ dom ({r} -◁ f). x + the (({r} -◁ f † [r + n ↦ the (f r) - n]) x) = r}
             = {x ∈ dom ({r} -◁ f). x + the (({r} -◁ f) x) = r}
              by (metis Int-iff ‹{r} -◁ f ∪m [r + n ↦ the (f r) - n] = {r} -◁ f † [r + n ↦ the
(f r) - n]›
                 disjoint-dom-antirestr dom-eq-singleton-conv empty-iff f'-restr the-dagger-dom-left)
            moreover have  {x ∈ dom [r + n ↦ the (f r) - n]. x + the (({r} -◁ f † [r + n ↦
the (f r) - n]) x) = r}
                  = {x ∈ dom [r + n ↦ the (f r) - n]. x + the ([r + n ↦ the (f r) - n] x) = r}
                   by (metis (lifting) l-dagger-apply)
           ultimately show ?thesis by auto
          qed
         qed
        qed
       also have ... = {}
         proof -
          have {x∈ dom ([r + n ↦ the (f r) - n]).
                x + the  ([r + n ↦ the (f r) - n] x) = r} = {}
           by (smt add-implies-diff comm-monoid-add-class.add.left-neutral
              diff-add-zero dom-empty empty-Collect-eq empty-iff gr-n fun-upd-same
              l-inmapupd-dom-iff less-nat-zero-code nat-add-commute the.simps)
          moreover have {x ∈ dom ({r} -◁ f).
```

```
            x + the ((({r} -◁ f) x) = r} = {}
            proof -
             have {x ∈ dom ({r} -◁ f). x + the ((({r} -◁ f) x) = r}
               ⊆ {x ∈ dom (f). x + the (f x) = r}
                 by (smt Collect-cong eq-refl f-in-dom-ar-the-subsume l-dom-ar-not-in-dom r-in-dom
sep-def sepf)
              also have ... = {}
               by (smt Collect-empty-eq r-in-dom sep-def sepf)
              finally show ?thesis by simp
             qed
            ultimately show ?thesis by simp
          qed
          finally show ?thesis by simp
        qed
       qed
      have above-shape: dispose1-above ({r} -◁ f ∪m [r + n ↦ the (f r) - n]) r n = [r + n ↦ the(f
r) - n]
      unfolding dispose1-above-def
      proof -
       have {x ∈ dom ({r} -◁ f ∪m [r + n ↦ the (f r) - n]). x = r + n} = {r+n}
       proof -
        have {x ∈ dom ({r} -◁ f ∪m [r + n ↦ the (f r) - n]). x = r + n} =
                {x ∈ dom ({r} -◁ f † [r + n ↦ the (f r) - n]). x = r + n}
                unfolding munion-def by (subst disjoint-dom-antirestr, simp)
        also have ... =  {x ∈ dom ({r} -◁ f). x = r + n} ∪
             {x ∈ dom ([r + n ↦ the (f r) - n]). x = r + n}
             by(subst l-dagger-dom,rule union-comp)
        also have ... = {x ∈ dom ({r} -◁ f). x = r + n} ∪ {r+n}
          by auto
        also have ... = {r+n}
        proof -
          have  {x ∈ dom ({r} -◁ f). x = r + n} = {}
         by (metis (lifting, mono-tags) Collect-empty-eq gr-n inv l-dom-ar-not-in-dom l-plus-s-not-in-f
nat1n r-in-dom)
          thus ?thesis by auto
        qed
        finally show ?thesis by simp
       qed
       moreover have {r+n} ◁ ({r} -◁ f ∪m [r + n ↦ the (f r) - n]) = [r + n ↦ the (f r) - n]
       proof (subst l-dom-r-singleton)
        show r + n ∈ dom ({r} -◁ f ∪m [r + n ↦ the (f r) - n])
          by (smt calculation empty-Collect-eq insert-compr)
        next
        show [r + n ↦ the ((({r} -◁ f ∪m [r + n ↦ the (f r) - n]) (r + n))]
             = [r + n ↦ the (f r) - n]
             by (metis dagger-upd-dist disjoint-dom-antirestr fun-upd-same munion-def the.simps)
       qed
       ultimately show  {x ∈ dom ({r} -◁ f ∪m [r + n ↦ the (f r) - n]). x = r + n} ◁ ({r} -◁ f
∪m [r + n ↦ the (f r) - n]) = [r + n ↦ the (f r) - n]
       by auto
      qed
      have min-loc-shape: min-loc (dispose1-ext ({r} -◁ f ∪m [r + n ↦ the (f r) - n]) r n) = r
       unfolding dispose1-ext-def
      proof ( simp add: above-shape below-shape)
       show min-loc ([r + n ↦ the (f r) - n] ∪m Map.empty ∪m [r ↦ n]) = r
       proof -
```

```
        have without-empty: [r + n ↦ the (f r) - n] ∪m Map.empty ∪m [r ↦ n]
        = [r + n ↦ the (f r) - n] ∪m [r ↦ n]
          by (metis l-munion-empty-rhs)
        then have min-loc ([r + n ↦ the (f r) - n] ∪m [r ↦ n])
              = min-loc ([r + n ↦ the (f r) - n] † [r ↦ n])
        proof -
         have dom ([r + n ↦ the (f r) - n])∩ dom( [r ↦ n]) = {r+n} ∩ {r}
           by auto
         also have ... = {} using nat1n by auto
         finally show ?thesis by (simp add: munion-def)
        qed
        also have ...  = min (min-loc [r + n ↦ the (f r) - n]) (min-loc [r ↦ n])
        by(rule min-loc-dagger,simp-all)
       also have ... = min (r+n) (r) by (simp add: min-loc-singleton)
       also have ... = r by simp
       finally show ?thesis using without-empty by simp
      qed
     qed
     have sum-size-shape: sum-size (dispose1-ext ({r} -⊲ f ∪m [r + n ↦ the (f r) - n]) r n) = the(f
r)
       unfolding dispose1-ext-def
     proof ( simp add: above-shape below-shape)
       show sum-size ([r + n ↦ the (f r) - n] ∪m empty ∪m [r ↦ n]) = the (f r)
       proof -
          have without-empty: [r + n ↦ the (f r) - n] ∪m Map.empty ∪m [r ↦ n]
          = [r + n ↦ the (f r) - n] ∪m [r ↦ n]
            by (metis l-munion-empty-rhs)
          then have sum-size ([r + n ↦ the (f r) - n] ∪m [r ↦ n])
              = sum-size ([r + n ↦ the (f r) - n]) + sum-size ([r ↦ n])
              apply (subst  sum-size-munion, simp-all)
              by (metis nat1-def nat1n)
          also have ... = the (f r) - n + n by (simp add: sum-size-singleton)
          also have ... = the (f r)  by (metis gr-n le-add-diff-inverse
                                    nat-add-commute termination-basic-simps(5))
          finally show ?thesis using without-empty by simp
        qed
      qed
      from d1-post show ?thesis
      proof (simp only: dispose1-equiv, unfold dispose1-post2-def)
      assume f'' = (dom (dispose1-below f' r) ∪ dom (dispose1-above f' r n)) -⊲ f'
         ∪m [min-loc (dispose1-ext f' r n) ↦ sum-size (dispose1-ext f' r n)]
      then have f'' =
            {r+n}  -⊲ f' ∪m [min-loc (dispose1-ext f' r n) ↦ sum-size (dispose1-ext f' r n)]
             by (simp add: f'-restr below-shape above-shape)
     also have ... = ({r} -⊲ f) ∪m [min-loc (dispose1-ext f' r n) ↦ sum-size (dispose1-ext f' r n)]
       proof -
         have {r+n} -⊲ f' = {r+n} -⊲ ({r} -⊲ f ∪m [r + n ↦ the (f r) - n])
           by (simp add: f'-restr)
         also have ... =  {r+n} -⊲ ({r} -⊲ (f ∪m [r + n ↦ the (f r) - n]))
         proof(subst l-munion-dom-ar-assoc)
           show  {r} ⊆ dom f by (simp add: r-in-dom)
         next
           show  dom f ∩ dom [r + n ↦ the (f r) - n] = {} by (rule disjoint-dom)
         next
           show {r + n} -⊲ {r} -⊲ (f ∪m [r + n ↦ the (f r) - n]) =
                {r + n} -⊲ {r} -⊲ (f ∪m [r + n ↦ the (f r) - n]) ..
```

```
        qed
        also have ... =  {r} -◁ ({r+n} -◁ (f ∪m [r + n ↦ the (f r) - n]))
          by (metis l-dom-ar-singletons-comm)
         also have ... =  {r} -◁ ({r+n} -◁ (f † [r + n ↦ the (f r) - n]))
          unfolding munion-def
          by (simp only: disjoint-dom,simp)
         also have ... =  {r} -◁ f
         proof (subst antirestr-then-dagger-notin)
          show r+n ∉ dom f using disjoint-dom by auto
          next
          show  {r} -◁ f = {r} -◁ f ..
         qed
         finally show ?thesis by simp
       qed
       also have ... =  ({r} -◁ f) ∪m [r ↦ the (f r)]
          by (simp add: min-loc-shape sum-size-shape f'-restr)
       also have ... =  ({r} -◁ f) † [r ↦ the (f r)]
        proof -
          have dom ({r} -◁ f) ∩ dom [r ↦ the (f r)] = {}
        by (metis Collect-conj-eq Collect-conv-if2 Collect-mem-eq dom-eq-singleton-conv f-in-dom-ar-notelem
inf-commute singleton-conv2)
          thus ?thesis by (simp add: munion-def)
        qed
     also have ... = f using r-in-dom by (rule antirestr-then-dagger)
     finally show ?thesis ..
   qed
  qed
qed
qed

end

theory HEAP01ReifyProofsIJW
imports HEAP01Reify HEAP1ProofsIJW
begin



lemma nat-nonzero-induct [case-names base step]:
  assumes base: P (1::nat)
  and grzero: x>0
  and step: ⋀(x::nat). x>0 ⟹ P x ⟹ P(x+1)
  shows P x
using assms
apply induct
sorry


lemma contig-nonabut-finite-set-induct [case-names empty extend, induct set: finite]:
  assumes fin: finite F
    and empty: P {}
    and extend: ⋀ F F'. finite F ⟹
                        finite F' ⟹
                        F' ≠ {} ⟹
                        contiguous F' ⟹
```

non-abut F F′ ⟹
(∗ *Part of abut-def* *F ∩ F′ = {} ⟹* ∗)
P F ⟹
P (F ∪ F′)
shows *P F*
sorry

definition
non-abut :: *nat set ⇒ nat set ⇒ bool*
where
non-abut s1 s2 ≡
disjoint s1 s2 ∧ (∗ *Nothing equal*! ∗)
(∀· *l1∈s1*. ∀·*l2∈s2*. (*l2 > l1+1*) ∨ (*l1 > l2+1*))

lemma *non-abut-commute*: *non-abut F F′ = non-abut F′ F*
unfolding *non-abut-def disjoint-def* **by** *auto*

lemma *non-abut-subset*: *non-abut F F′ ⟹ Fsub ⊆ F ⟹ F′sub ⊆ F′*
⟹ non-abut Fsub F′sub
unfolding *non-abut-def disjoint-def* **apply** *auto*
apply (*erule-tac x=l1* **in** *ballE*)
apply (*erule-tac x=l2* **in** *ballE*)
apply *simp*
by *auto*

lemma (**in** *level1-basic*) *fin-retrieve*: *finite* (*retr0*(*f1*))
proof -
from *l1-invariant-def* **have** *finf1*: *finite* (*dom f1*)
by (*metis invF1-finite-weaken*)
from *l1-invariant-def* **have** (*nat1-map f1*)
by (*metis invF1-nat1-map-weaken*)
thus *?thesis* **unfolding** *retr0-def locs-def*
apply *simp*
apply (*simp add*: *finf1*)
apply (*rule ballI*)
apply (*rule locs-of-finite*)
apply (*simp add*: *nat1-map-def*)
done
qed

lemma *non-abut-sep*:
assumes *non-abutting*: ∀· *l* ∈ *dom f*. ∀· *l′* ∈ *dom f*. *l≠l′* ⟶ *non-abut* (*locs-of l* (*the* (*f l*)))
(*locs-of l′* (*the* (*f l′*)))
and *nat1f*: *nat1-map f*
shows *sep f*
unfolding *sep-def*
proof

```
 fix l
 assume lindom: l ∈ dom f
 show  l + the (f l) ∉ dom f
 proof -
  have l + the (f l) - 1 ∈ locs-of l (the (f l))
  proof (subst b-locs-of-as-set-interval)
   show  nat1 (the (f l)) by (metis ‹l ∈ dom f› nat1-map-def nat1f)
  next
   show (l + the (f l)) - 1 ∈ {l..<l + the (f l)}
    by (metis ‹l ∈ dom f› b-locs-of-as-set-interval nat1-map-def nat1f top-locs-of)
  qed
  have dom f ≠ {} by (metis ‹l ∈ dom f› empty-iff)
  have flg0: the (f l) > 0 by (metis ‹l ∈ dom f› nat1-map-def nat1f nat1-def)
  show ?thesis
  proof (cases dom f = {l})
   assume dom f = {l}
   then show l + the (f l) ∉ dom f
    using flg0 by simp
  next
   assume dom f ≠ {l}
   show ?thesis
   proof
    assume ∗:l + the (f l) ∈ dom f
    then have ∃· l' ∈ dom f. l ≠ l'
     by (metis add-diff-cancel-left' comm-monoid-diff-class.diff-cancel flg0 gr-implies-not0)
    then obtain l' where l'indom: l' ∈ dom f and l'eq: l' = l + the (f l) and noteq: l≠l'
     using ∗ by (metis add-0-iff flg0 less-not-refl)
    then have non-abut
                (locs-of l                (the (f    l)))
                (locs-of (l + (the (f l)))  (the (f ( (l + (the (f l)))))))
              by (metis lindom non-abutting)
     then have non-abut-rhs: (∀·l1∈locs-of l (the (f l)). ∀·l2∈locs-of (l + the (f l)) (the (f (l +
the (f l))))).
         l1 + 1 < l2 ∨ l2 + 1 < l1) unfolding non-abut-def by simp

    obtain l1 where l1locs: l1 ∈ locs-of l (the (f l))

                    and l1shape: l1 = l + (the (f l)) - 1

              by (metis ‹l + the (f l) - 1 ∈ locs-of l (the (f l))›)
    obtain l2 where l2locs: l2∈locs-of (l + the (f l)) (the (f (l + the (f l))))
           and l2shape: l2 = l + the (f l)
       using ∗ by (auto simp: l-dom-in-locs-of nat1f)
    from non-abut-rhs have (l+ the (f l) - 1) + 1 < l + the (f l) ∨ l + the (f l) + 1 < (l+ the
(f l) - 1)
    using l1locs apply (erule-tac x=l1 in ballE)
    using l2locs apply (erule-tac x=l2 in ballE)
     by (simp-all add: l1shape l2shape)
    then show False by auto
   qed
  qed
 qed
qed

lemma non-abut-Disjoint:
 assumes non-abutting: ∀· l ∈ dom f. ∀· l' ∈ dom f. l≠l' ⟶ non-abut (locs-of l (the (f l)))
```

```
      (locs-of l' (the (f l')))
   shows Disjoint f
   unfolding Disjoint-def
proof(intro ballI impI)
  fix l l'
  assume lindom: l ∈ dom f and l'indom: l' ∈ dom f and noteq: l ≠ l'
  show disjoint (Locs-of f l) (Locs-of f l')
  proof -
    from non-abutting
    have non-abut (locs-of l (the (f l))) (locs-of l' (the (f l')))
      using lindom l'indom noteq by (auto)
    then show ?thesis unfolding non-abut-def Locs-of-def
     by (simp add: lindom l'indom)
  qed
qed



lemma non-abut-VDM-inv:
 assumes non-abutting: ∀· l ∈ dom f. ∀· l' ∈ dom f. l≠l' ⟶ non-abut (locs-of l (the (f l)))
      (locs-of l' (the (f l')))
  and nat1f: nat1-map f
 shows VDM-F1-inv f
 unfolding VDM-F1-inv-def
  by (metis nat1f non-abut-Disjoint non-abut-sep non-abutting)



lemma min-contig:
fixes m l :: nat
assumes atleastone: l>0
shows Min {i::nat. m ≤ i ∧ i < m + l} = m
proof (induct l rule: nat-nonzero-induct)
  have {i::nat. m ≤ i ∧ i < m + (1::nat)} = {m} by auto
  then show Min {i. m ≤ i ∧ i < m + 1} = m by simp
  next
  show 0 < l by (rule atleastone)
 next
  fix x
  assume ∗: 0<x
  and  ind-hyp: Min {i::nat. m ≤ i ∧ i < m + x} = m
  show Min {i::nat. m ≤ i ∧ i < (m + (x+  1))} = m
  proof -
    have ∗∗: {i::nat. m ≤ i ∧ i < (m + (x + 1))} = {i::nat. m ≤ i ∧ i < m + x} ∪ {m+x}
      by auto
    then have Min {i::nat. m ≤ i ∧ i < Suc (m + x)} =
     Min ({i::nat. m ≤ i ∧ i < m + x} ∪ {m+x}) by auto
    also have ...=  min (m+x) (Min {i::nat. m ≤ i ∧ i < m + x})
     by (subst Min-insert [symmetric], auto simp add: ∗)
    also have ...= min (m+x) m using ind-hyp by auto
    finally show ?thesis using ∗ by auto
  qed
qed
```

lemma *card-contig*: *card* $\{i::nat.\ m \leq i \wedge i < m + l\} = l$
proof (*induct l*)
show *base*: *card* $\{i::nat.\ m \leq i \wedge i < m + (0::nat)\} = (0::nat)$
by *simp*
next
fix *l*
assume *ind-hyp*: *card* $\{i::nat.\ m \leq i \wedge i < m + l\} = l$
show *card* $\{i::nat.\ m \leq i \wedge i < m + Suc\ l\} = Suc\ l$
proof -
have $\{i::nat.\ m \leq i \wedge i < m + Suc\ l\} = \{i::nat.\ m \leq i \wedge i < m + l\} \cup \{m+l\}$
by *auto*
from *this ind-hyp* **show** *?thesis* **by** *auto*
qed
qed

lemma *retr0-empty*: *retr0 empty* $= \{\}$
unfolding *retr0-def locs-def nat1-map-def*
by *auto*

lemma *empty-retr0*: *nat1-map x* $\Longrightarrow$ *retr0 x* $= \{\} \Longrightarrow x =$ *empty*
unfolding *retr0-def locs-def* **apply** *simp*
by (*metis empty-iff k-in-locs-iff l-map-non-empty-has-elem-conv*
not-dom-not-locs-weaken)

lemma *mapdom-in-retr*: *x*$\in$*dom f* $\Longrightarrow$ *the* (*f x*) $> 0 \Longrightarrow$
nat1-map f $\Longrightarrow x \in$ (*retr0 f*)
unfolding *retr0-def locs-def locs-of-def* **by** *auto*

lemma *non-empty-nat1-card*: *finite F* $\Longrightarrow F \neq \{\} \Longrightarrow$ *card F* > 0
by *auto*

lemma *eq-locs*:
assumes *finF*: *finite F'*
and *nonempF*: $F' \neq \{\}$
and *contig*: *contiguous F'*
shows *locs-of* (*Min F'*) (*card F'*) $= F'$
proof -
from *finF nonempF* **have** *nat1* (*card F'*) **unfolding** *nat1-def*
by (*rule non-empty-nat1-card*)
from *contig* **obtain** *m l* **where** *F'shape*: $F' =$ *locs-of m l* **and** *lgrzero*: $l > 0$
unfolding *contiguous-def* **by** *auto*
then have $m =$ *Min* (*F'*)
by (*simp add*: *locs-of-def F'shape min-contig lgrzero*)
moreover have $l =$ *card F'*
by (*simp add*: *F'shape locs-of-def lgrzero card-contig*)
ultimately show *?thesis* **using** *F'shape* **by** *blast*
qed

```
lemma (in level0-basic) free1-adequacy:
shows ∃·! f1. (f0 = retr0 f1 ∧ F1-inv f1)
proof -

  from  l0-invariant-def have finf0: finite f0 by (metis F0-inv-defs)
  from l0-input-notempty-def have nat1s1: nat1 s0 by metis
  from finf0 show ?thesis
  proof (induct rule: contig-nonabut-finite-set-induct)
    case empty

    show ∃·!f1. {} = retr0 f1 ∧ F1-inv f1
    proof(rule-tac a=empty in ex1I, rule conjI)
      show {} = retr0 Map.empty  by (simp only: retr0-empty)
     next
      show F1-inv Map.empty  by (simp only: F1-inv-empty)
     next
      fix x
      assume {} = retr0 x ∧ F1-inv x
      then show x = empty
        by (metis empty-retr0 invF1-nat1-map-weaken)
    qed
   next
    fix F F'
    assume F'-finite: finite F'
    and notemp: F' ≠ {}
    and F'-contig: contiguous F'
    and F'-nonabut: non-abut F F'
    and exist-hyp: ∃·!f1. F = retr0 f1 ∧ F1-inv f1
    from exist-hyp obtain f1hook
    where ind-hyp-retr: F = retr0 f1hook
     and ind-hyp-inv: F1-inv f1hook
     and ind-hyp-nat1: nat1-map f1hook by auto
   show ∃·!f1. F ∪ F' = retr0 f1 ∧ F1-inv f1
   proof(rule-tac a=(f1hook ∪m [Min(F') ↦ card F']) in ex1I, rule conjI)

     have nonzerorange: ∀·l ∈ dom f1hook. (the (f1hook l)) > 0
        by (metis nat1-def nat1-map-def ind-hyp-nat1)
      have non-intersect: F ∩ F' = {}
         by (metis F'-nonabut non-abut-def disjoint-def)
      have domsubsetretr: dom f1hook ⊆ retr0 f1hook
      proof
        fix x assume indom: x ∈ dom f1hook
        then show x ∈ retr0 f1hook
        proof (rule mapdom-in-retr)
          show 0 < (the (f1hook x))
             using nonzerorange indom by auto
         next
          show nat1-map f1hook by (rule ind-hyp-nat1)
        qed
      qed
     then have subsetF: dom f1hook ⊆ F using ind-hyp-retr by auto
     have min-notin-f1hook: Min F' ∉ dom f1hook
     proof -
       have Min F' ∈ F'
```

```
      using notemp F'-finite by auto
    then have *: Min F' ∉ F
      using non-intersect by auto
    have Min F' ∉ dom (f1hook) using subsetF * by auto
    thus ?thesis by simp
  qed
  have nat1-card-F': nat1 (card F')
    by (metis F'-finite bot-less bot-nat-def card-eq-0-iff nat1-def notemp)
  show F ∪ F' = retr0 (f1hook ∪m [Min F' ↦ card F'])


  proof -
    have F ∪ F' = retr0 (f1hook ∪m [Min(F') ↦ card F'])
    proof -
      have dom-extend: dom (f1hook ∪m [Min(F') ↦ card F'])
                     = insert (Min(F')) (dom f1hook)
        by (simp add: l-munion-dom min-notin-f1hook)
      have nat1-upd-state: nat1-map (f1hook ∪m [Min(F') ↦ card F'])
      by(rule unionm-singleton-nat1-map,
            simp-all add: ind-hyp-nat1 min-notin-f1hook nat1-card-F' del: nat1-def)
      then have retr0 (f1hook ∪m [Min(F') ↦ card F'])
           = (⋃ s::nat∈dom (f1hook ∪m [Min F' ↦ card F']).
          locs-of s (the ((f1hook ∪m [Min F' ↦ card F']) s)))
        unfolding retr0-def locs-def by simp
      also have (⋃ s::nat∈dom (f1hook ∪m [Min F' ↦ card F']).
                   locs-of s (the ((f1hook ∪m [Min F' ↦ card F']) s)))
          = (⋃ s::nat∈insert (Min F') (dom f1hook).
                   locs-of s (the ((f1hook ∪m [Min F' ↦ card F']) s)))
       by (simp only: dom-extend)
      also have ... = locs-of (Min F') (the ((f1hook ∪m [Min F' ↦ card F']) (Min F')))
              ∪ (⋃ s::nat∈dom f1hook. locs-of s (the ((f1hook ∪m [Min F' ↦ card F']) s)))
        by (simp only: UN-insert)
      also have ... = F' ∪ (⋃ s::nat∈dom f1hook. locs-of s (the ((f1hook ∪m [Min F' ↦ card F'])
s)))

      proof -

        have F' = locs-of (Min F') (card F')
         by (metis F'-contig F'-finite eq-locs notemp)
        moreover have the ((f1hook ∪m [Min F' ↦ card F']) (Min F')) = card F'
        apply (subst l-munion-apply)
        apply (simp add: min-notin-f1hook)
        by simp
        ultimately show ?thesis by auto
      qed
      also have ... =(⋃ s::nat∈dom f1hook. locs-of s (the ((f1hook ∪m [Min F' ↦ card F']) s))) ∪
F'
       by blast
      also have ... = F ∪ F'
      proof -
        have (⋃ s∈dom f1hook. locs-of s (the ((f1hook ∪m [Min F' ↦ card F']) s))) = F
        proof -
          have ∀ ·s ∈ dom f1hook.
          locs-of s (the (f1hook s)) =
          locs-of s (the ((f1hook ∪m [Min F' ↦ card F']) s))
          apply (subst l-munion-apply)
```

```
        apply (metis Int-insert-right-if0 dom-eq-singleton-conv inf-bot-right min-notin-f1hook)
        by (smt domIff l-inmapupd-dom-iff min-notin-f1hook)

      thus ?thesis using ind-hyp-retr retr0-def locs-def nat1-upd-state ind-hyp-nat1
       by auto
    qed
    thus ?thesis by simp
  qed
  finally show ?thesis ..
 qed
 thus ?thesis .
qed

from ind-hyp-inv show upd-inv: F1-inv (f1hook ∪m [Min F' ↦ card F'])
proof
 assume ind-sep: sep f1hook and ind-Disjoint: Disjoint f1hook
 and ind-finite: finite (dom (f1hook)) and ind-nat1-map: nat1-map f1hook
 show ?thesis
 proof(rule invF1-shape)

  from min-notin-f1hook ind-hyp-nat1 nat1-card-F'
   show nat1-ext: nat1-map (f1hook ∪m [Min F' ↦ card F'])
   by (rule unionm-singleton-nat1-map)

  from min-notin-f1hook ind-finite show finite (dom (f1hook ∪m [Min F' ↦ card F']))
   by (rule unionm-singleton-finite )

 show VDM-F1-inv (f1hook ∪m [Min F' ↦ card F'])
 proof
  from min-notin-f1hook ind-sep show sep (f1hook ∪m [Min F' ↦ card F'])
   proof (rule unionm-singleton-sep)
    show ∀ ·l∈dom f1hook. l + the (f1hook l) ∉ dom [Min F' ↦ card F']
    proof
     fix l assume lindom: l ∈ dom f1hook
     show l + the (f1hook l) ∉ dom [Min F' ↦ card F']
     proof
      assume ∗: l + the (f1hook l) ∈ dom [Min F' ↦ card F']
      have l + (the (f1hook l)) = Min F'
       by (metis ∗ l-inmapupd-dom-iff l-map-non-empty-has-elem-conv)
      then have l + (the (f1hook l)) ∈ F'
       by (metis F'-finite Min-in notemp)
      obtain l1 where l1shape: l1 = l + (the (f1hook l)) - 1 by simp
      have l1inF: l1 ∈ F apply (subst ind-hyp-retr)
      unfolding retr0-def locs-def using ind-nat1-map apply simp
      apply (rule-tac x=l in bexI)
      apply (metis ‹l1 = l + the (f1hook l) - 1› lindom nat1-map-def top-locs-of)
      apply (rule lindom)
      done
      obtain l2 where l2shape: l2 = l + the (f1hook l) by simp
      have l2inF': l2 ∈ F'
          by (metis ‹l + the (f1hook l) ∈ F'› ‹l2 = l + the (f1hook l)›)
      from F'-nonabut have contra: l1 + 1 < l2 ∨ l2 + 1 < l1
       unfolding non-abut-def using l1inF l2inF' by simp
      from contra l1shape l2shape show False
       by auto
     qed
```

```
        qed
       show Min F' + card F' ∉ dom f1hook
        proof
          assume ∗: Min F' + card F' ∈ dom f1hook
          have Min F' + card F' - 1 ∈ F'
                by (metis F'-contig all-not-in-conv contiguous-def eq-locs locs-of-finite nat1-card-F'
top-locs-of)
          obtain l1 where l1shape: l1 = Min F' + card F' by simp
          have l1inF: l1 ∈ F by (metis ∗ ⟨l1 = Min F' + card F'⟩ set-mp subsetF)
          obtain l2 where l2shape: l2 = Min F' + card F' - 1 by simp
          have l2inF': l2 ∈ F' by (metis ⟨Min F' + card F' - 1 ∈ F'⟩ ⟨l2 = Min F' + card F' - 1⟩)
          from F'-nonabut have contra: l1 + 1 < l2 ∨ l2 + 1 < l1
          unfolding non-abut-def using l1inF l2inF' by simp
         from contra l1shape l2shape show False by auto
         qed
      next
       show nat1 (card F') by (rule nat1-card-F')
    qed
      from min-notin-f1hook ind-Disjoint ind-nat1-map
      show Disjoint (f1hook ∪m [Min F' ↦ card F'])
     proof (rule unionm-singleton-Disjoint)
       show nat1 (card F') by (rule nat1-card-F')
       next
       show disjoint (locs-of (Min F') (card F')) (locs f1hook)
       by (metis F'-contig F'-finite disjoint-def eq-locs ind-hyp-retr
           inf-commute non-intersect notemp retr0-def)
     qed
    qed
   qed
  qed

  fix x assume ∗: F ∪ F' = retr0 x ∧ F1-inv x
  show x = f1hook ∪m [Min F' ↦ card F']
  proof (rule locs-unique)
    show locs x = locs (f1hook ∪m [Min F' ↦ card F'])
      by (metis ∗
          ⟨F ∪ F' = retr0 (f1hook ∪m [Min F' ↦ card F'])⟩ retr0-def)
   next
    show F1-inv x using ∗ by simp
   next
    show F1-inv (f1hook ∪m [Min F' ↦ card F']) by (rule upd-inv)
   next
    show x ≠ Map.empty
          by (metis (full-types) ∗ empty-subsetI notemp retr0-empty
              sup.right-idem sup-absorb1 sup-commute)
   next
    show f1hook ∪m [Min F' ↦ card F'] ≠ Map.empty
      by (metis l-munion-singleton-not-empty min-notin-f1hook)
  qed
 qed
qed
qed
```

theorem *r-free01-widen-pre*:
PO-l01-new-widen-pre
unfolding *PO-l01-new-widen-pre-def*
new1-pre-def new0-pre-def
proof(*intro allI conjI impI*, *elim conjE exE*)
fix *f1 s1 l*
assume *invf1*: *F1-inv f1*
and *nat1s1*: *nat1 s1*
and *new0pre*: *is-block l s1* (*retr0 f1*)
have *locs-subset*: *locs-of l s1* ⊆ *locs f1*
by (*metis is-block-def new0pre retr0-def*)
moreover have *l* ∈ *locs-of l s1* **using** *nat1s1*
by (*simp add*: *b-locs-of-as-set-interval*)
ultimately have *l* ∈ *locs f1* **by** *auto*
then have *l* ∈ (⋃*s*∈*dom f1*. *locs-of s* (*the* (*f1 s*)))
unfolding *locs-def Locs-of-def*
by (*simp add*: *invf1 invF1-nat1-map-weaken*)
from *locs-subset invf1 nat1s1*
have ∃· *m*∈ *dom f1*. *locs-of l s1* ⊆ *locs-of m* (*the* (*f1 m*))
by (*rule locs-locs-of-subset*)
then obtain *m* **where** *mindom*: *m*∈*dom f1* **and**
locssubm: *locs-of l s1* ⊆ *locs-of m* (*the* (*f1 m*))
by *auto*
then have *mgrs1*: *s1* ≤ *the* (*f1 m*)
proof (*cases l*=*m*)
assume *l* = *m*
then have *locs-of l s1* ⊆ *locs-of l* (*the* (*f1 l*)) **by** (*metis locssubm*)
show *?thesis*
proof (*rule locs-of-subset-range*)
show *0* < *s1* **by** (*metis nat1-def nat1s1*)
show *0* < *the* (*f1 m*) **by** (*metis invF1-sep-weaken comm-monoid-add-class.add.right-neutral*
invf1 mindom neq0-conv sep-def)
show *locs-of l s1* ⊆ *locs-of l* (*the* (*f1 m*))
by (*metis* ‹*l* = *m*› ‹*locs-of l s1* ⊆ *locs-of l* (*the* (*f1 l*))›)
qed
next
assume *lnotm*: *l* ≠ *m*
have *m* < *l*
proof(*rule ccontr*)
assume ¬ *m* < *l*
then have ∗: *m* > *l* **by** (*metis lnotm nat-neq-iff*)
have *l* ∉ *locs-of m* (*the* (*f1 m*))
apply (*rule less-a-not-in-locs-of*)
apply (*metis invF1-sep-weaken comm-monoid-add-class.add.right-neutral invf1 mindom neq0-conv*
sep-def)
by (*simp add*: ∗)
thus *False* **by** (*metis* ‹*l* ∈ *locs-of l s1*› *locssubm set-mp*)
qed
show *?thesis*
proof (*rule locs-of-subset-range-gr*)
show *0* < *s1* **by** (*metis nat1-def nat1s1*)
show *0* < *the* (*f1 m*) **by** (*metis invF1-sep-weaken comm-monoid-add-class.add.right-neutral*
invf1 mindom neq0-conv sep-def)
show *locs-of l s1* ⊆ *locs-of m* (*the* (*f1 m*))
by (*metis locssubm*)
show *m* < *l* **by** (*metis* ‹*m* < *l*›)

```
    qed
  qed
  thus ∃·l∈dom f1. s1 ≤ the (f1 l)
   by (metis mindom)
qed



lemma strangesets: B ⊆ A ⟹ D ⊆ B ⟹ (A - B) ∪ (B-D) = A-D
by auto

theorem r-free01-narrow-post:
  assumes invf1: F1-inv f1
  and invf1': F1-inv f1'
  and nat1s1: nat1 s1
  and new0pre: new0-pre (retr0 f1) s1
  and new1post: new1-post f1 s1 f1' r
  shows new0-post (retr0 f1) s1 (retr0 f1') r
proof -
  from  new0pre new1post show ?thesis
   unfolding new0-post-def new1-post-def new0-pre-def new1-post-eq-def new1-post-gr-def retr0-def
  proof(elim disjE exE conjE, intro conjI)
    assume f1r: the (f1 r) = s1 and rindom: r ∈ dom f1
    show is-block r s1 (locs f1)
     unfolding is-block-def
    proof
      show nat1 s1 by (rule nat1s1)
     next
      show locs-of r s1 ⊆ locs f1
       by (metis invF1-nat1-map-weaken rindom f1r invf1 l-locs-of-within-locs)
    qed
   next
    assume ∗: f1' = {r} -◁ f1  and rindom: r ∈ dom f1 and f1r: the (f1 r) = s1
    show locs f1' = locs f1 - locs-of r s1
    proof (subst ∗, subst dom-ar-locs)
      show finite (dom f1) by (metis invF1-finite-weaken invf1)
     next
      show  nat1-map f1 by (metis invF1-nat1-map-weaken invf1)
     next
      show Disjoint f1  by (metis invF1-Disjoint-weaken invf1)
     next
      show r ∈ dom f1 by (rule rindom)
     next
      show locs f1 - locs-of r (the (f1 r)) = locs f1 - locs-of r s1
       by (simp add: f1r)
    qed
   next
    fix l
    assume  isblockl: is-block l s1 (locs f1)
          and rindom: r ∈ dom f1
          and s1less: s1 < the (f1 r)
          and f1'shape: f1' = {r} -◁ f1 ∪m [r + s1 ↦ the (f1 r) - s1]
    show is-block r s1 (locs f1) ∧ locs f1' = locs f1 - locs-of r s1
    proof
      show is-block r s1 (locs f1)
       unfolding is-block-def
```

```
    proof
      show nat1 s1 by (rule nat1s1)
     next
      show  locs-of r s1 ⊆ locs f1
      proof -
        have  locs-of r s1 ⊆ locs-of r (the (f1 r))
          by (metis Diff-subset less-trans locs-of-minus nat1-def nat1s1 s1less)
        moreover have locs-of r (the (f1 r)) ⊆ locs f1
          by (metis invF1-nat1-map-weaken ‹r ∈ dom f1› invf1 l-locs-of-within-locs)
        ultimately show ?thesis by simp
      qed
    qed
   next
    show locs f1' = locs f1 - locs-of r s1
    proof (subst f1'shape, subst locs-unionm-singleton)
      show nat1 (the (f1 r) - s1) by (metis diff-is-0-eq nat1-def neq0-conv not-le s1less)
     next
      show nat1-map ({r} -◁ f1) by (metis invF1-nat1-map-weaken dom-ar-nat1-map invf1)
     next
      show  r + s1 ∉ dom ({r} -◁ f1) by (metis invf1 l-dom-ar-notin-dom-or l-plus-s-not-in-f nat1s1
rindom s1less)
     next
      show  locs ({r} -◁ f1) ∪ locs-of (r + s1) (the (f1 r) - s1) = locs f1 - locs-of r s1
      proof (subst dom-ar-locs)
        show finite (dom f1) by (metis invF1-finite-weaken invf1)
       next
        show  nat1-map f1 by (metis invF1-nat1-map-weaken invf1)
       next
        show Disjoint f1  by (metis invF1-Disjoint-weaken invf1)
       next
        show r ∈ dom f1 by (rule rindom)
       next
       show locs f1 - locs-of r (the (f1 r)) ∪ locs-of (r + s1) (the (f1 r) - s1) = locs f1 - locs-of r s1
        proof -
          have  locs-of r s1 =  locs-of r (the (f1 r)) - locs-of (r+s1) ((the (f1 r)) - s1)
           by (metis add-0-iff invf1 l-plus-s-not-in-f less-trans locs-of-minus nat1s1 neq0-conv rindom
s1less)
          have **:  locs-of (r+s1) ((the (f1 r)) - s1) = locs-of r (the (f1 r)) -  locs-of r s1
          by (metis ‹locs-of r s1 = locs-of r (the (f1 r)) - locs-of (r + s1) (the (f1 r) - s1)› double-diff
locs-of-subset nat1-def s1less subset-refl zero-less-diff)
          show ?thesis
          proof (subst **,subst strangesets)
            show locs-of r (the (f1 r)) ⊆ locs f1
              by (metis invF1-nat1-map-weaken invf1 l-locs-of-within-locs rindom)
           next
            show  locs-of r s1 ⊆ locs-of r (the (f1 r))
             by (metis Diff-subset ‹locs-of r s1 = locs-of r (the (f1 r)) - locs-of (r + s1) (the (f1 r) -
s1)›)
           next
            show  locs f1 - locs-of r s1 = locs f1 - locs-of r s1
            by (rule refl)
          qed
        qed
      qed
    qed
  qed
```

```
  qed
qed

lemma PO-l01-new-narrow-post
  by (metis PO-l01-new-narrow-post-def r-free01-narrow-post)

lemma g2-subset: B ⊆ A ⟹ (A - B) ∪ (B ∪ C) = A ∪ C by auto
lemma g3-lemma: Y ⊆ X ⟹ Z ⊆ X ⟹ (X - Y - Z) ∪ (Z ∪ Y ∪ A) = X ∪ A
  by auto


theorem r-free01-dispose-widen-pre:
    PO-l01-dispose-widen-pre
    unfolding PO-l01-dispose-widen-pre-def
              dispose1-pre-def
              disjoint-def retr0-def dispose0-pre-def
by simp


lemma (in level1-dispose) r-free01-dispose-narrow-post:
  assumes invf1: F1-inv f1
  and invf1': F1-inv f1'
  and nat1s1: nat1 s1
  and dis0pre: dispose0-pre (retr0 f1) d1 s1
  and dis1post: dispose1-post2 f1 d1 s1 f1'
  shows dispose0-post (retr0 f1) d1 s1 (retr0 f1')
proof -
  from invf1 show ?thesis
  proof
    assume sepf1: sep f1 and disjf1: Disjoint f1
    and nat1f1: nat1-map f1 and finitef1: finite (dom f1)
    from invf1' show ?thesis
    proof
      assume sepf1': sep f1' and disjf1': Disjoint f1'
      and nat1f1': nat1-map f1' and finitef1': finite (dom f1')
      from dis1post show ?thesis
        unfolding dispose0-post-def
                  dispose1-post2-def
      proof -
        assume *: f1' = (dom (dispose1-below f1 d1) ∪ dom (dispose1-above f1 d1 s1)) -◁ f1 ∪m
          [min-loc (dispose1-ext f1 d1 s1) ↦ HEAP1.sum-size (dispose1-ext f1 d1 s1)]
        show retr0 f1' = retr0 f1 ∪ locs-of d1 s1
        proof (subst *)
          from dis0pre have locs-of d1 s1 ∩ retr0 f1 = {} by (metis dispose0-pre-def)
          then have d1notf1: d1 ∉ dom f1
           by (metis IntI empty-iff l-locs-of-within-locs locs-of-extended
            nat1-map-def nat1f1 nat1s1 retr0-def set-rev-mp top-locs-of top-locs-of2)
          show retr0 ((dom (dispose1-below f1 d1) ∪ dom (dispose1-above f1 d1 s1)) -◁ f1 ∪m
              [min-loc (dispose1-ext f1 d1 s1) ↦ HEAP1.sum-size (dispose1-ext f1 d1 s1)]) =
              retr0 f1 ∪ locs-of d1 s1
          proof (cases dispose1-below f1 d1 = empty)
            assume belowempty: dispose1-below f1 d1 = Map.empty
            show ?thesis
            proof (cases dispose1-above f1 d1 s1 = empty)
              assume aboveempty: dispose1-above f1 d1 s1 = Map.empty
              then show ?thesis          unfolding dispose1-ext-def retr0-def
```

```
                proof (simp add: aboveempty belowempty l-munion-empty-rhs l-munion-empty-lhs
l-dom-ar-empty-lhs min-loc-singleton sum-size-singleton)
              show locs (f1 ∪m [d1 ↦ s1]) = locs f1 ∪ locs-of d1 s1
              proof (rule locs-unionm-singleton)
                show nat1 s1 by (metis nat1s1)
                show nat1-map f1 by (metis nat1f1)
                show d1 ∉ dom f1 by (rule d1notf1)
              qed
             qed
            next
             assume abovenotempty: dispose1-above f1 d1 s1 ≠ Map.empty
              have abovebelowshape: (dom (dispose1-below f1 d1) ∪ dom (dispose1-above f1 d1 s1)) =
{d1+s1}
                by (simp add: belowempty l-munion-empty-rhs l-munion-empty-lhs l-dom-ar-empty-lhs
d1notf1
                 d1-not-dispose-above d1-not-dispose-below above-dom abovenotempty)
              have min-loc-shape: min-loc (dispose1-ext f1 d1 s1) = d1
               by (metis belowempty l-map-non-empty-dom-conv min-below-empty)
              have sum-size (dispose1-ext f1 d1 s1) = sum-size (dispose1-above f1 d1 s1) + s1
               by (simp add: dispose1-ext-def belowempty l-munion-empty-rhs
                 l-munion-empty-lhs l-dom-ar-empty-lhs d1notf1
                 d1-not-dispose-above d1-not-dispose-below sum-size-munion
                 finite-dispose1-above abovenotempty sum-size-singleton)
              then have sum-size-shape: sum-size (dispose1-ext f1 d1 s1) = the(f1 (d1+s1)) + s1
               by (simp add: above-sumsize abovenotempty)
              show ?thesis unfolding retr0-def
              proof (simp add: sum-size-shape min-loc-shape abovebelowshape)
                show locs ({d1 + s1} -⊲ f1 ∪m [d1 ↦ the (f1 (d1 + s1)) + s1]) = locs f1 ∪ locs-of
d1 s1
               proof (subst locs-unionm-singleton)
                 show nat1 (the (f1 (d1 + s1)) + s1) by (metis nat1-dispose1-ext sum-size-shape)
                next
                 show nat1-map ({d1 + s1} -⊲ f1) by (metis dom-ar-nat1-map nat1f1)
                next
                 show d1 ∉ dom ({d1 + s1} -⊲ f1) by (metis d1notinf1 l-dom-ar-notin-dom-or)
                next
                  show locs ({d1 + s1} -⊲ f1) ∪ locs-of d1 (the (f1 (d1 + s1)) + s1) = locs f1 ∪
locs-of d1 s1
                 proof (subst dom-ar-locs, rule finitef1,rule nat1f1,rule disjf1)
                   show d1 + s1 ∈ dom f1 by (metis above-d1s1-in-f1 abovenotempty)
                  next
                   show locs f1 - locs-of (d1 + s1) (the (f1 (d1 + s1))) ∪ locs-of d1 (the (f1 (d1 +
s1)) + s1)
                         = locs f1 ∪ locs-of d1 s1
                   proof -
                     have locs-of (d1 + s1) (the (f1 (d1 + s1))) ⊆ locs f1 by (metis above-d1s1-in-f1
abovenotempty k-in-locs-iff nat1f1 subsetI)
                    moreover have locs-of d1 (the (f1 (d1 + s1)) + s1) =
                      locs-of (d1 + s1) (the (f1 (d1 + s1))) ∪ locs-of d1 s1
                    proof -
                      have locs-of d1 (the (f1 (d1 + s1)) + s1) = locs-of d1 (s1 + the (f1 (d1 +
s1)))
                       by (metis nat-add-commute)
                      also have ... = locs-of d1 s1 ∪ locs-of (d1 + s1) (the (f1 (d1 + s1)))
                       by (metis above-d1s1-in-f1 abovenotempty locs-of-sum-range nat1-map-def nat1f1
nat1s1)
```

```
            finally show ?thesis by auto
          qed
          ultimately show ?thesis by (simp add: g2-subset)
        qed
      qed
    qed
  qed
qed
next
 assume belownotempty: dispose1-below f1 d1 ≠ Map.empty
 from belownotempty have ∃·x. x∈dom f1 ∧ x + the (f1 x) = d1
 proof -
  have dispose1-below f1 d1 ≠ empty by (rule belownotempty)
  then have { x ∈ dom f1 . x + the(f1 x) = d1 } ≠ {}
   by (metis (full-types) dispose1-below-def l-dom-r-nothing)
  thus ?thesis  by (smt empty-Collect-eq)
 qed
 then obtain below where belowinf1: below∈dom f1
       and belowplusf1below: below + the (f1 below) = d1
     by metis
 then have below-in-dom: below ∈ dom(dispose1-below f1 d1)
  unfolding dispose1-below-def
 proof (subst l-dom-r-iff)
  show below ∈ {x ∈ dom f1. x + the (f1 x) = d1} ∩ dom f1
   by (smt Int-Collect belowinf1 belowplusf1below inf-commute)
 qed
 have below-shape: dispose1-below f1 d1 = [below ↦ the (f1 below)]
 proof
  fix x
  show dispose1-below f1 d1 x = [below ↦ the (f1 below)] x
  proof (simp, intro allI impI conjI)
   from below-in-dom
   show dispose1-below f1 d1 below = Some (the (f1 below))
    unfolding dispose1-below-def
   proof (subst f-in-dom-r-apply-the-elem)
    show below ∈ dom f1 by (rule belowinf1)
   next
    show below ∈ {x ∈ dom f1. x + the (f1 x) = d1}
     by (smt belowinf1 belowplusf1below mem-Collect-eq)
   qed(rule refl)
  next
   assume xnoteqbelow: x ≠ below
   show dispose1-below f1 d1 x = None
   proof(rule ccontr)
    assume dispose1-below f1 d1 x ≠ None then
    have con: x ∈ dom (dispose1-below f1 d1)
     by auto
    from con have xindomrset: x ∈ {x ∈ dom f1. x + the (f1 x) = d1}
     unfolding dispose1-below-def
     by (metis (full-types) l-in-dom-dom-r)
    then have xinf: x∈ dom f1 by (simp add: xindomrset)
    have xeqd1: x + the (f1 x) = d1
     by (metis (lifting, mono-tags) mem-Collect-eq xindomrset)

    from disjf1 have *: locs-of x (the (f1 x)) ∩  locs-of below (the (f1 below)) = {}
     by (metis xnoteqbelow belowinf1 Disjoint-def
```

disjoint-def l-locs-of-Locs-of-iff xinf)
have *nat1below*: *nat1* (*the* (*f1 below*)) **by** (*metis nat1-map-def nat1f1 belowinf1*)
have *nat1x*: *nat1* (*the* (*f1 x*)) **by** (*metis nat1-map-def nat1f1 xinf*)
from *xinf xeqd1 belowplusf1below belowinf1 nat1x nat1below*
have ∗∗: *locs-of x* (*the* (*f1 x*)) ∩ *locs-of below* (*the* (*f1 below*)) ≠ {}
by (*metis IntI ex-in-conv top-locs-of*)
from ∗ ∗∗ **show** *False* **by** *simp*
qed
qed
qed
then have *dom-below*: *dom* (*dispose1-below f1 d1*) = {*below*} **by** *simp*
have *sum-size-below*: *sum-size* (*dispose1-below f1 d1*) = *the* (*f1 below*)
by (*simp add*: *sum-size-singleton below-shape*)
show *?thesis*
proof (*cases dispose1-above f1 d1 s1* = *empty*)
assume *aboveempty*: *dispose1-above f1 d1 s1* = *Map.empty*
have *abovebelow-shape*: (*dom* (*dispose1-below f1 d1*)) ∪ (*dom* (*dispose1-above f1 d1 s1*))
= {*below*}
by (*simp add*: *aboveempty dom-below*)
have *min-loc-shape*: *min-loc* (*dispose1-ext f1 d1 s1*) = *below*
by (*metis dom-below insert-not-empty min-below-notempty singleton-iff*)
have *sum-size-shape*: *sum-size* (*dispose1-ext f1 d1 s1*) = *the* (*f1 below*) + *s1*
unfolding *dispose1-ext-def*
by (*simp add*: *aboveempty l-munion-empty-lhs sum-size-munion sum-size-singleton*
finite-dispose1-below belownotempty d1-not-dispose-below sum-size-below)
show *?thesis* **unfolding** *retr0-def*
proof (*simp add*: *sum-size-shape min-loc-shape abovebelow-shape*)
show *locs* ({*below*} -◁ *f1* ∪*m* [*below* ↦ *the* (*f1 below*) + *s1*]) = *locs f1* ∪ *locs-of d1 s1*
proof (*subst locs-unionm-singleton*)
show *nat1* (*the* (*f1 below*) + *s1*) **by** (*metis nat1-dispose1-ext sum-size-shape*)
next
show *nat1-map* ({*below*} -◁ *f1*) **by** (*metis dom-ar-nat1-map nat1f1*)
next
show *below* ∉ *dom* ({*below*} -◁ *f1*) **by** (*metis f-in-dom-ar-notelem*)
next
show *locs* ({*below*} -◁ *f1*) ∪ *locs-of below* (*the* (*f1 below*) + *s1*) = *locs f1* ∪ *locs-of d1 s1*
proof (*subst dom-ar-locs*, *rule finitef1*, *rule nat1f1*, *rule disjf1*)
show *below* ∈ *dom f1* **by** (*metis belowinf1*)
next
show *locs f1 - locs-of below* (*the* (*f1 below*)) ∪ *locs-of below* (*the* (*f1 below*) + *s1*) = *locs f1* ∪ *locs-of d1 s1*
proof -
have *locs-of below* (*the* (*f1 below*)) ⊆ *locs f1*
by (*metis belowinf1 l-locs-of-within-locs nat1f1*)
moreover have ∗: *below* + *the* (*f1 below*) = *d1* **by** (*metis belowplusf1below*)
moreover have *locs-of below* (*the* (*f1 below*) + *s1*) = *locs-of below* (*the* (*f1 below*)) ∪ *locs-of d1 s1*
proof -
have *locs-of below* ((*the* (*f1 below*)) + *s1*) = (*locs-of below* (*the* (*f1 below*))) ∪ (*locs-of* (*below* + (*the* (*f1 below*)))) *s1*
by (*metis locs-of-sum-range belowinf1 nat1-map-def nat1f1 nat1s1*)
also have ... = (*locs-of below* (*the* (*f1 below*))) ∪ *locs-of d1 s1*
by (*simp add*: ∗)
finally show *?thesis* .
qed
ultimately show *?thesis* **by** (*simp add*: *g2-subset*)

```
            qed
          qed
        qed
      qed
    next
      assume abovenotempty: dispose1-above f1 d1 s1 ≠ Map.empty
      have above-below-shape: (dom (dispose1-below f1 d1) ∪ dom (dispose1-above f1 d1 s1))
                = {below,d1+s1}
        by (metis Un-insert-left above-dom abovenotempty dom-below sup-bot-left)
      have min-loc-shape: min-loc (dispose1-ext f1 d1 s1) = below
        by (metis dom-below insert-not-empty min-below-notempty singleton-iff)
      have sum-size-shape: sum-size (dispose1-ext f1 d1 s1)
                    =  the (f1 (d1 + s1)) + the (f1 below) + s1
      proof -
        have sum-size-above-below: sum-size (dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1)
                  =  the (f1 (d1 + s1)) + the (f1 below)
         by (simp add: sum-size-munion finite-dispose1-above finite-dispose1-below abovenotempty
           belownotempty above-sumsize sum-size-below)
        then show ?thesis unfolding dispose1-ext-def
        proof (subst sum-size-munion)
          show finite (dom (dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1)) and
            finite (dom [d1 ↦ s1])
          by (simp-all add: finite-dispose1-above finite-dispose1-below k-finite-munion)
         next
          show dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1 ≠ empty
           and [d1 ↦ s1] ≠ empty
           by (auto simp: munion-notempty-right belownotempty)
         next
          from d1-not-above-below show dom (dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1) ∩
dom [d1 ↦ s1] = {}
           by simp
         next
          show sum-size (dispose1-above f1 d1 s1 ∪m dispose1-below f1 d1) + sum-size [d1 ↦ s1]
               = the (f1 (d1 + s1)) + the (f1 below) + s1
            by (simp add: sum-size-above-below sum-size-singleton)
        qed
      qed
      show ?thesis unfolding retr0-def
    proof (simp add: sum-size-shape min-loc-shape above-below-shape, subst locs-unionm-singleton)
          show nat1 (the (f1 (d1 + s1)) + the (f1 below) + s1) by (metis nat1-dispose1-ext
sum-size-shape)
        next
         show nat1-map ({below, d1 + s1} -◁ f1) by (metis dom-ar-nat1-map nat1f1)
        next
         show below ∉ dom ({below, d1 + s1} -◁ f1) by (metis insertI1 l-dom-ar-notin-dom-or)
        next
         show locs ({below, d1 + s1} -◁ f1) ∪ locs-of below (the (f1 (d1 + s1)) + the (f1 below) +
s1) = locs f1 ∪ locs-of d1 s1
         proof -
          have ∗: (locs ({below, d1 + s1} -◁ f1)) = locs ( {below} -◁ ({d1 + s1} -◁ f1))
           by (metis Un-empty-left Un-insert-left l-dom-ar-accum)
          show ?thesis
           proof (subst ∗, subst dom-ar-locs)
            show finite (dom ({d1 + s1} -◁ f1)) by (metis finite-Diff finitef1 l-dom-dom-ar)
           next
            show nat1-map ({d1 + s1} -◁ f1) by (metis dom-ar-nat1-map nat1f1)
```

next
show *Disjoint* ({*d1* + *s1*} -◁ *f1*) **by** (*metis disjf1 dom-ar-Disjoint*)
next
show *below* ∈ *dom* ({*d1* + *s1*} -◁ *f1*)
by (*metis* (*mono-tags*) *after-locs-of-not-in-locs belowinf1 belowplusf1below inf.commute inf-nat-def l1-invariant-def l-in-dom-ar nat1f1 nat-min-absorb1 not-dom-not-locs-weaken singletonE*)
next
show *locs* ({*d1* + *s1*} -◁ *f1*) - *locs-of below* (*the* (({*d1* + *s1*} -◁ *f1*) *below*)) ∪
locs-of below (*the* (*f1* (*d1* + *s1*)) + *the* (*f1 below*) + *s1*)
= *locs f1* ∪ *locs-of d1 s1*
proof (*subst dom-ar-locs*, *rule finitef1*, *rule nat1f1*, *rule disjf1*)
show *d1* + *s1* ∈ *dom f1* **by** (*metis above-d1s1-in-f1 abovenotempty*)
next

have ∗: *the* (({*d1* + *s1*} -◁ *f1*) *below*) = *the* (*f1 below*)
by (*metis belowplusf1below d1notinf1 domIff dom-antirestr-def inf.commute inf-nat-def nat-min-absorb1 singletonE*)
show *locs f1* - *locs-of* (*d1* + *s1*) (*the* (*f1* (*d1* + *s1*)))
- *locs-of below* (*the* (({*d1* + *s1*} -◁ *f1*) *below*))
∪ *locs-of below* (*the* (*f1* (*d1* + *s1*)) + *the* (*f1 below*) + *s1*)
= *locs f1* ∪ *locs-of d1 s1*
proof(*subst* ∗)
have *locs-of below* (*the* (*f1* (*d1* + *s1*)) + *the* (*f1 below*) + *s1*)
= *locs-of below* (*the* (*f1 below*) + (*the* (*f1* (*d1* + *s1*)) + *s1*))
by (*metis nat-add-commute nat-add-left-commute*)
also have ... = (*locs-of below* (*the* (*f1 below*))) ∪
(*locs-of* (*below* + (*the* (*f1 below*))) (*the* (*f1* (*d1* + *s1*)) + *s1*))
apply (*subst locs-of-sum-range*)
apply (*metis belowinf1 nat1-map-def nat1f1*)
apply (*simp add*: *nat1s1*)
apply (*rule disjI2*)
apply (*metis nat1-def nat1s1*)
by *simp*
also have ... = (*locs-of below* (*the* (*f1 below*)))
∪ (*locs-of d1* (*s1* + *the* (*f1* (*d1* + *s1*))))
by (*metis belowplusf1below nat-add-commute*)
also have ... = (*locs-of below* (*the* (*f1 below*)))
∪ (*locs-of d1 s1*) ∪ (*locs-of* (*d1*+*s1*) (*the* (*f1* (*d1* + *s1*))))
apply (*subst locs-of-sum-range*)
apply (*metis nat1s1*)
apply (*metis above-d1s1-in-f1 abovenotempty nat1-map-def nat1f1*)
by *auto*
finally have ∗∗∗: *locs-of below* (*the* (*f1* (*d1* + *s1*)) + *the* (*f1 below*) + *s1*)
= (*locs-of below* (*the* (*f1 below*)))
∪ (*locs-of* (*d1*+*s1*) (*the* (*f1* (*d1* + *s1*)))) ∪ (*locs-of d1 s1*)
by *auto*
show *locs f1* - *locs-of* (*d1* + *s1*) (*the* (*f1* (*d1* + *s1*))) - *locs-of below* (*the* (*f1 below*)) ∪ *locs-of below* (*the* (*f1* (*d1* + *s1*)) + *the* (*f1 below*) + *s1*) =
locs f1 ∪ *locs-of d1 s1*
proof (*subst* ∗∗∗)
show *locs f1* - *locs-of* (*d1* + *s1*) (*the* (*f1* (*d1* + *s1*))) - *locs-of below* (*the* (*f1 below*)) ∪
(*locs-of below* (*the* (*f1 below*)) ∪
locs-of (*d1* + *s1*) (*the* (*f1* (*d1* + *s1*))) ∪ *locs-of d1 s1*) =
locs f1 ∪ *locs-of d1 s1*

proof(*subst g3-lemma*)
show *locs-of* (*d1* + *s1*) (*the* (*f1* (*d1* + *s1*))) ⊆ *locs f1* **by** (*metis above-d1s1-in-f1 abovenotempty l-locs-of-within-locs nat1f1*)
next
show *locs-of below* (*the* (*f1 below*)) ⊆ *locs f1* **by** (*metis belowinf1 l-locs-of-within-locs nat1f1*)
next
show *locs f1* ∪ *locs-of d1 s1* = *locs f1* ∪ *locs-of d1 s1* **by** (*rule refl*)
qed
qed
qed
qed
qed
qed
qed
qed
qed
qed
qed
qed

end

Appendix G

Earlier Heap models using ZEves

In this chapter we refer to the two technical reports / documents about the AI$_4$FM first attempts at the heap problem. They are available through the AI$_4$FM website[1] and include three versions of the Z/EVES models. The versions can be found at these links below.

- Z/EVES heap level 0 and 1 (v0) = TR-ZEVES-heapL0L1-v0.pdf
- Z/EVES heap level 0 and 1 (v1) = TR-ZEVES-heapL0L1-v1.pdf
- Z/EVES heap level 0 and 1 (v2) = TR-ZEVES-heapL0L1-v2.pdf

[1] http://www.ai4fm.org/tr

Bibliography

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[Abr10] J.-R. Abrial. *The Event-B Book*. Cambridge University Press, Cambridge, UK, 2010.

[AD10] Serge Autexier and Dominik Dietrich. A tactic language for declarative proofs. In Matt Kaufmann and LawrenceC. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 99–114. Springer Berlin Heidelberg, 2010.

[BBHI05a] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[BBHI05b] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.

[BCJ84] H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BFW09] Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Science of Computer Programming*, 74(4):219–237, 2009. cited By (since 1996) 10.

[BN09] J.C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *TAP*, 2009.

[CB08] David Copper and Janet Barnes. Tokeneer ID station EAL5 demonstrator: Summary report. Technical Report S.P1229.81.1 Issue: 1.1, Altran-Praxis, August 2008.

[DEP12a] DEPLOY. Enhanced deployment in the automotive sector (WP1). EU Project Deliverable D38, DEPLOY Project, 2012.

[DEP12b] DEPLOY. Enhanced deployment in the space sector (WP3). EU Project Deliverable D39, DEPLOY Project, 2012.

[Fre04] Leo Freitas. Proving theorems with z/eves. Technical report, University of York, 2004.

[FW08] Leo Freitas and Jim Woodcock. Mechanising mondex with z/eves. *Formal Aspects of Computing*, 20(1):117–139, 2008. cited By (since 1996) 11.

[FW09] Leo Freitas and Jim Woodcock. A chain datatype in z. *International Journal of Software and Informatics*, 3(2-3):357–374, 2009.

[GKL13] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. A graphical language for proof strategies. To appear in LPAR'13. Also available at arXiv:1302.6890, 2013.

[Gro12] Gudmund Grov. A graphical strategy language for proof re-use, 2012. slides (from Oxford?).

[Har96] John Harrison. Proof style. In *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172. Springer-Verlag, 1996.

[HK13] Jonathan Heras and Ekaterina Komendantskaya. Ml4pg in computer algebra verication. In *Conferences on Intelligent Computer Mathematics*, 2013.

[JFV13] Cliff B. Jones, Leo Freitas, and Andrius Velykis. Ours is to reason why. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods*, volume 8051 of *LNCS*, pages 227–243, 2013.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[JLS12] C. B. Jones, M. J. Lovert, and L. J. Steggles. A semantic analysis of logics that cope with partial functions. In John Derrick et al., editors, *ABZ 2012*, volume 7316 of *Lecture Notes in Computer Science*, pages 252–265, June 2012.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon95] C.B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.

[JOW06] Cliff Jones, Peter O'Hearn, and Jim Woodcock. Verified software: a grand challenge. *IEEE Computer*, 39(4):93–95, 2006.

[JS90] C. B. Jones and R. C. F. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990.

[KMM09] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *ACL2 Computer-Aided Reasoning: An Approach*. University of Austin Texas, 2009.

[MU05] Petra Malik and Mark Utting. CZT: A framework for Z tools. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB*, volume 3455 of *LNCS*, pages 65–84. Springer, 2005.

[Nau72] Peter Naur. An experiment on program development. *BIT*, 12:347–365, 1972.

[NPW02a] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002.

[NPW02b] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, May 2002.

[P+94] Lawrence C Paulson et al. *The Isabelle reference manual*. Citeseer, 1994.

[Pau94] Lawrence C. Paulson. Isabelle: A Generic Theorem Prover. *Lecture Notes in Computer Science*, 828, 1994.

[PB10] Lawrence C Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. *IWIL-2010*, 2010.

[Saa97] Mark Saaltink. The Z/EVES system. In Jonathan Bowen, Michael Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer Berlin / Heidelberg, 1997.

[Saa99] Mark Saaltink. Z/eves 2.0 user's guide. Technical Report TR-99-5493-06a, ORA Canada, 1999.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, second edition, December 2008.

[Sch12] Matthias Schmalz. *Formalising the Logic of Event-B*. PhD thesis, ETH, Zuerich, 2012.

[Sle13] Nataliia Sleshina. Proof engineering through a proof process. Master's thesis, School of Computing Science, Newcastle University, 2013.

[Vel12] Andrius Velykis. Inferring the proof process. In Christine Choppy, David Delayahe, and Kaïs Klaï, editors, *FM2012 Doctoral Symposium*, Paris, France, August 2012.

[Vel14] Andrius Velykis. *Capturing & Inferring the Proof Process (under submission)*. PhD thesis, School of Computing Science, Newcastle University, 2014.

[WD96] Jim Woodcock and Jim Davies. *Using Z*. Prentice Hall International, 1996.

[Wen02] Markus M. Wenzel. *Isabelle/Isar - a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2002.

[WF08] Jim Woodcock and Leo Freitas. Linking VDM and Z. In *International Conference on Engineering of Complex Computer Systems*, pages 143–152, Belfast, 2008. cited By (since 1996) 0; Conference of 13th IEEE International Conference on the Engineering of Complex Computer Systems, ICECCS 2008; Conference Date: 31 March 2008 through 4 April 2008; Conference Code: 72055.

[Whi13] Iain Whiteside. *Refactoring Proofs*. PhD thesis, School of Informatics, August 2013.

[Wie02] Freek Wiedijk. Formal proof sketches. In *In Proceedings of TYPES'03, volume 3085 of LNCS*, pages 378–393. Springer, 2002.

[WWW13] WWW. AI4FM, as at February 2013. `http://www.ai4fm.org`.