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

1 Introduction ................................................. 5  
1.1 Inferring proof intent .................................. 6  
1.2 Proof engineering ........................................ 7  
1.3 Proof obligations in formal methods ....................... 8  
1.4 Outline ................................................ 8  

2 Modelling heap storage .............................. 10  
2.1 Heap as a set of (contiguous) locations (level 0) ................. 10  
2.2 Heap as a disjoint map of location sizes (level 1) ............... 11  
2.3 Feasibility ................................................ 13  
2.4 Sanity checks ............................................. 14  
2.5 Reification POs ............................................ 15  

3 Models of why .............................................. 16  
3.1 Bodies of knowledge ...................................... 18  
3.1.1 Base theories (as Body objects) ....................... 18  
3.1.2 Specifications give rise to bodies ....................... 18  
3.2 Proof objects ............................................. 19  
3.2.1 Conjectures ............................................ 19  
3.2.2 Justifications .......................................... 20  
3.2.3 Representing a hand proof .............................. 21  
3.3 Strategies ................................................ 24  
3.3.1 Data .................................................... 24  
3.3.2 Selecting strategies .................................... 25  
3.3.3 Facing lemma gaps ..................................... 28  
3.3.4 Capturing strategies ................................... 28  
3.3.5 An analogy (formally known as “Leo’s 2c worth”) .......... 29  
3.4 Relations between bodies .................................. 29  
3.5 Summary of “Model” ...................................... 30  
3.5.1 Data structure .......................................... 30  
3.5.2 Discussion of the model ................................. 31  

4 How we got to where we are ...................... 33  
4.1 Models of a heap: VDM originals ......................... 34  
4.1.1 Heap as a set of locations (L0) ....................... 34  
4.1.2 Heap as a set of pieces (L1) ........................... 35  
4.2 Theorem proving experiences ............................. 36  
4.2.1 Z/EVES v0 — warts and all .......................... 36  
4.2.2 Z/EVES v1 — interface and postcondition change ......... 37  
4.2.3 Z/EVES v2 — invariant packaging and abstraction .......... 40
CONTENTS

D VDM Maps auxiliary library  
D.1 Extra map operators .................................................. 89
D.2 Set operators lemmas .................................................. 90
D.3 Map operators lemmas .................................................. 90

E Heap lemmas and proofs (Leo)  
E.1 HEAP0 Isabelle (automation) lemmas ................................. 103
E.1.1 locs_of weakening lemmas [EXPERT] ............................ 103
E.2 Feasibility proof obligations for HEAP level 0 .................... 103
E.3 Proof of some properties of interest ............................... 104
E.3.1 Invariant ............................................................ 104
E.3.2 Operations .......................................................... 104
E.4 General Lemmas ........................................................ 106
E.5 Goal-oriented - invariant update .................................... 112
E.6 Goal-oriented - DISPOSE1 invariant update ...................... 118
E.7 NEW 1 proofs .......................................................... 128
E.7.1 NEW 1 FSB ......................................................... 129
E.8 DISPOSE 1 proofs ..................................................... 131
E.9 Proof of some properties of interest ............................... 135
E.9.1 Invariant testing ................................................... 135
E.9.2 Operations properties .............................................. 136

F Heap lemmas and proofs (Iain)  

G Earlier Heap models using ZEves  212
Chapter 1

Introduction

In a UK-funded project known as AI 4FM\(^1\) 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](http://www.ai4fm.org)

\(^2\)See [http://www.deploy-project.eu/](http://www.deploy-project.eu/)
1.1. INFERRING PROOF INTENT

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\footnote{See http://www.overturetool.org}. 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
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 “Whys” 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 (i.e., 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.

\(1.2 \text{ Proof engineering}\)

We call the streamlining of such proof processes \textbf{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 (\textit{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

\[^4\text{See}\ \text{http://www.computing.dundee.ac.uk/staff/katya/MLAPG/}\]
1.3. PROOF OBLIGATIONS IN FORMAL METHODS

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; etc5.

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

---

5This list is not exhaustive, but those that came up during the development involving Tokeneer’s abstract specification [CB08].
CHAPTER 1. INTRODUCTION

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 AI4FM 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 \texttt{locs-of}, which we use the Isabelle \texttt{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:

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

The issue of handling adjacency is handled by accepting that Loc is synonymous 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:

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

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

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

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

$$is-block(l, n, ls) \triangleq locs-of(l, n) \subseteq ls$$
CHAPTER 2. MODELLING HEAP STORAGE

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.

**NEW**0 \((s; N_1) \rightarrow \text{Loc}
\)

**ext wr** \(f_0 : \text{Free}0\)

**pre** \(\exists l \in \text{Loc} \cdot \text{is-block}(l, s, f_0)\)

**post** \(\text{is-block}(l, s f_0) \land f_0 = f_0 - \text{locs-of}(r, s)\)

**DISPOSE**0 \((l; \text{Loc}, s; N_1)\)

**ext wr** \(f_0 : \text{Free}0\)

**pre** \(\text{locs-of}(l, s) \cap f_0 = \{\}\)

**post** \(f_0 = f_0 \cup \text{locs-of}(l, s)\)

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).

\(\text{is-disj} : X\text{-set} \times X\text{-set} \rightarrow \mathbb{B}\)

\(\text{is-disj}(s, t) \triangleq s \cap t = \{\}\)

\(\text{Free}1 = \text{Loc} m \rightarrow N_1\)

\(\text{inv} (f) \triangleq\)

\(\forall l, l' \in \text{dom} f \cdot\)

\(l \neq l' \Rightarrow \text{is-disj(locs-of}(l, f(l)), \text{locs-of}(l', f(l')))) \land\)

\(\forall l \in \text{dom} f \cdot (l + f(l)) \notin \text{dom} f\)

The definition of **NEW**1 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 (\(\preceq\)) 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.

\[
\text{NEW1} \ (s; N_1) \rightharpoonup \text{Loc}
\]

\[
\text{ext} \ w r \ f_1 : \text{Free1}
\]

\[
\text{pre} \ \exists l \in \text{dom} f_1 \cdot f_1(l) \geq s
\]

\[
\text{post} \ r \in \text{dom} f_1 \land
\begin{align*}
(f_1(r) &= s \land f_1 = \{r\} \triangleleft f_1 \\
(f_1(r) &> s \land f_1 = (\{r\} \triangleleft f_1) \cup \{r + s \mapsto f_1(r) - s\})
\end{align*}
\]

The complexity of ordering non-abutting locations is brought to the surface in the definition of both \textit{NEW1} and \textit{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 extended 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.

\[
\text{DISPOSE1} \ (d; \text{Loc}, s; N_1)
\]

\[
\text{ext} \ w r \ f : \text{Free1}
\]

\[
\text{pre} \ \text{is-disj} ( \text{locs-of}(d, s), \text{locs}(f))
\]

\[
\text{post} \ \exists \text{below}, \text{above}, \text{ext} \in \text{Loc} \to N_1 :
\begin{align*}
\text{below} &= \{l \mid l \in \text{dom} f \land l + f(l) = d\} \triangleleft f \land \\
\text{above} &= \{l \mid l \in \text{dom} f \land l = d + s\} \triangleleft f \land \\
\text{ext} &= \text{above} \cup \text{below} \cup \{d \mapsto s\} \land \\
f &= (\text{dom} \text{below} \cup \text{dom} \text{above} \triangleleft f) \cup \{\text{min-loc}(\text{ext}) \mapsto \text{sum-size}(\text{ext})\}
\end{align*}
\]

where:

\[
\text{min-loc} : (\text{Loc} \to N_1) \to \text{Loc}
\]

\[
\text{min-loc}(sm) \triangleq \text{if} \ sm = \{x \mapsto y\} \text{ then } x \text{ else let } x \in \text{dom} sm \text{ in } \text{min}(x, \text{min-loc}(\{x \mapsto sm\})
\]

\[
\text{pre} \ sm \neq \{\}
\]

\[
\text{min} : N \times N \to N
\]

\[
\text{min}(x, y) \triangleq \text{if} \ x < y \text{ then } x \text{ else } y
\]

\[
\text{sum-size} : (\text{Loc} \to N_1) \to N
\]

\[
\text{sum-size}(sm) \triangleq \text{if} \ sm = \{\} \text{ then } 0 \text{ else let } x \in \text{dom} sm \text{ in } sm(x) + \text{sum-size}(\{x \mapsto sm\})
\]

Comments on model and proof On the strictly greater case for \textit{NEW1}, we make a design decision to choose the right-most section of contiguous memory to be the one allocated (\textit{i.e.} the maplet update as \{r + s \mapsto f_0(r)-s\}). We could have also defined a (more abstract) non-deterministic choice among any of the possible contiguous set of locations, which would lead \textit{NEW1} to be the exact inverse of \textit{DISPOSE1}. We chose a specific implementation without
CHAPTER 2. MODELLING HEAP STORAGE

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 DISPOSE\(^1\), 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 straight-
forward (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 NEW\(^1\).

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
NEW\(^1\) 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:

\[
\begin{align*}
\text{NEW0-feas} & \quad \exists f \in \text{Free}_0, r \in \text{Loc} \cdot \text{post-NEW0}(s, f_0, f, r) \\
\text{pre-NEW0}(f_0, s) & \quad f_0 \in \text{Free}_0, s \in \mathbb{N}_1
\end{align*}
\]

where the theorem stated in a prover looks like this:

\[
\forall s \in \mathbb{N}_1, f_0 \in \text{Free}_0 \cdot \text{pre-NEW0}(s, f_0) \Rightarrow \\
\exists r \in \text{Loc-set}, f_0 \in \text{Free}_0 \cdot \text{post-NEW0}(s, f_0, f, r)
\]

which is equivalent to

\[
\forall s \in \mathbb{N}_1, f_0 \in \text{Free}_0 \cdot \exists l \in \text{Loc}^* \cdot \text{is-block}(l, s, f_0) \Rightarrow \\
\exists r \in \text{Loc-set}, f_0 \in \text{Free}_0 \cdot \exists s \in \text{Loc}^* \cdot (\text{is-block}(r, s, f_0) \wedge \\
f_0 = f_0 - \text{locs-of}(r, s)
\]

For given inputs \((s)\) and before state \((f_0)\), 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 DISPOSE\(^0\) is similar:

\[
\begin{align*}
\text{DISPOSE0-feas} & \quad \exists f \in \text{Free}_0 \cdot \text{post-DISPOSE0}(d, s, f_0) \\
\text{pre-DISPOSE0}(d, s, f_0) & \quad f_0 \in \text{Free}_0, d \in \text{Loc}, s \in \mathbb{N}_1
\end{align*}
\]

\(^1\)This design decision also appears in the widen-precondition proof of the reification between levels 0 and 1
in Section 6.3.2
2.4. SANITY CHECKS

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:

\[
\begin{align*}
\overleftarrow{f_1} & \in Free_1, s \in N_1 \\
pre-NEW_1(s, \overleftarrow{f_1}) & \implies \exists f_1 \in Free_1, r \in Loc \cdot post-NEW_1(s, \overleftarrow{f_1}, f_1, r)
\end{align*}
\]

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

\[
\begin{align*}
\overleftarrow{f}, f & \in Free_0, r \in Loc, s \in N_1 \\
pre-NEW_0(\overleftarrow{f}, s) & \implies \exists f \in Free_0, r \in Loc \cdot post-NEW_0(\overleftarrow{f}, s, f, r)
\end{align*}
\]

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

\[
\begin{align*}
\overleftarrow{f}, f & \in Free_0, r \in Loc, s \in N_1 \\
pre-NEW_0(\overleftarrow{f}, s) & \implies \exists f_1, f_2 \in Free_0, r_1, r_2 \in Loc \cdot post-NEW_0(f_1, \overleftarrow{f}, s, f_2, r_1, r_2)
\end{align*}
\]

Another property could relate recently freed heap space:

\[
\begin{align*}
\overleftarrow{f}, f & \in Free_0, l \in Loc, s_1, s_2 \in N_1 \\
pre-DISPOSE_0(\overleftarrow{f}, l, s_1) & \implies \exists f, s_1 \geq s_2 \cdot post-DISPOSE_0(f, \overleftarrow{f}, l, s_1)
\end{align*}
\]

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).
CHAPTER 2. MODELLING HEAP STORAGE

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):

\[
\text{retr}_0 : \text{Free}1 \rightarrow \text{Free}0
\]

\[
\text{retr}_0(f) \triangleq \text{locs}(f)
\]

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:

\[
\text{Free}1\text{-adequacy} \quad \begin{aligned}
         & f_0 \in \text{Free}0 \\
         \exists f_1 \in \text{Free}1 : f_0 = \text{retr}_0(f_1)
\end{aligned}
\]

The second type of proof obligation is widen precondition, which states that (for \text{DISPOSE}):

\[\begin{aligned}
\text{DISPOSE}1\text{-w-pre} & \quad f \in \text{Free}1, l \in \text{Loc}, s \in \mathbb{N}_1 \\
& \quad \text{pre-DISPOSE0}(\text{retr}_0(f), l, s) \\
& \quad \text{pre-DISPOSE1}(f, l, s)
\end{aligned}\]

The third type is narrow postcondition, which (for \text{DISPOSE}) states:

\[\begin{aligned}
\text{DISPOSE}1\text{-n-post} & \quad f \in \text{Free}1, l \in \text{Loc}, s \in \mathbb{N}_1 \\
& \quad \text{pre-DISPOSE0}(\text{retr}_0(\overset{\rightarrow}{f}), l, s) \\
& \quad \text{post-DISPOSE1}(\overset{\rightarrow}{f}, l, s, f) \\
& \quad \text{post-DISPOSE0}(\text{retr}_0(\overset{\rightarrow}{f}), l, s, \text{retr}(f))
\end{aligned}\]
Chapter 3

Models of why

This chapter enlarges on [JFV13] in motivating and describing an abstract model of the AI4FM 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 AI4FM. 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.
CHAPTER 3. MODELS OF WHY

Figure 3.1: AI4FM “capture” mode

Figure 3.2: AI4FM “replay” mode
3.1. BODIES OF KNOWLEDGE

The conjectures and proof content of a body (Section 3.2) is fairly routine; Section 3.3 on
"strategies" is central to the realisation of the AI 4FM 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 4FM instantiation can be thought of as a collection of
bodies (in the sense of "body of knowledge").

$$\Sigma :: \ bdm : BdId \rightarrow Body$$

There will be bodies of knowledge about mathematical theories such as set theory (cf. Sec-
tion 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 Sec-
tion 3.4.)

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

- Body :: uses : BdId-\text{set}
- functions : FnId \rightarrow FnDefn

\text{FnDefn} :: type : \text{Signature}
- defn : [Definition]

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 \text{N} for the result of \text{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.)

$$\text{disj} : \text{Loc-set} \times \text{Loc-set} \rightarrow \text{B}$$

$$\text{disj}(s1, s2) \triangleq s1 \cap s2 = \{\}$$

This signature and definition are stored in a \text{FnDefn}.

Similarly, there would be a body (of knowledge) for $\text{Loc} \rightarrow \text{N}_1$ (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 specifi-
cation 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 refications 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 \text{inv-Free}1 and the
functions \text{all-locs and locs-of}.\footnote{For the purposes of this chapter, there has been some refactoring [Whi13] wrt Chapter 2: the function \text{locs}
has been renamed \text{all-locs} \text{d1 has been renamed} \text{d etc. This is particularly important for Section 3.2.3.}}

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:

\text{18}
would give rise to constructor and selector functions:

\[ mk - X : T_1 \times T_2 \rightarrow X \]
\[ f_1 : X \rightarrow T_1 \]
\[ f_2 : X \rightarrow T_2 \]

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.

\[
\text{Body} :: \cdots
\]
\[
guts : \text{ConjId} \xrightarrow{m} \text{Conjecture}
\]
\[
\cdots
\]

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:

\[
\text{Conjecture} :: \text{hyps} : \text{Judgement}^* \\
\text{goal} : \text{Judgement} \\
\text{status} : \{\text{Lemma, RewriteL2R, NegativeProperty, \cdots}\} \\
\text{justifs} : \text{JustId} \xrightarrow{m} (\text{Axiom} | \text{Trusted} | \text{Justification}) \\
\cdots
\]

*Judgement* = Typing | Sequent | Equation | Ordering | \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 \( E_1 \lor E_2 \) 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 \( S_1, S_2, S_3 : X\text{-set} \), \( S_1 \subseteq S_2 \) 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.
3.2. PROOF OBJECTS

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

\[
\begin{align*}
L1 & \quad \text{s}_1, \text{s}_2: X\text{-set} \\
& \quad \text{disj}(\text{s}_1, \text{s}_2) \\
& \quad \text{s}_3 \subseteq \text{s}_2 \\
& \quad \text{disj}(\text{s}_1, \text{s}_3) \\
L1.5 & \quad \text{s}_1, \text{s}_2: X\text{-set} \\
& \quad \text{disj}(\text{s}_2, \text{s}_1 - \text{s}_2) \\
L2 & \quad \text{disj}(\text{s}_1, \text{s}_3) \\
& \quad \text{disj}(\text{s}_2, \text{s}_3) \\
& \quad \text{disj}((\text{s}_1 \cup \text{s}_2), \text{s}_3)
\end{align*}
\]

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 \text{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 NEW0-feas and DISPOSE0-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 HEAP1 would be:

\[
\begin{align*}
\overset{f_1}{\text{f}} & \in \text{Free1, } d \in \text{Loc, } s \in \mathbb{N}_1 \\
\text{pre-Dispose1}(d, s, \overset{f_1}{f}) \\
\text{DISPOSE1-feas} & \exists f_1 \in \text{Free1} \cdot \text{post-DISPOSE1}(d, s, \overset{f_1}{f}, f_1)
\end{align*}
\]

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. Conjectures can have a mapping to different Justifications). 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,

\[
\begin{align*}
\text{Justification} & \quad \text{claim} & : (\text{ConjId} \mid \text{ToolOP}) \\
& \quad \text{subst} & : \text{Term} \xrightarrow{m} \text{Term} \\
& \quad \text{sub-probs} & : \text{ConjId-set}
\end{align*}
\]
CHAPTER 3. MODELS OF WHY

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”:

\[
\begin{align*}
\land\text{-El} & \quad \frac{E_1 \land E_2}{E_1} \\
\end{align*}
\]

(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; \(\lor\text{-}E\) is the obvious example. One nice property of \(\lor\text{-}E\) 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 \(\forall\text{Needed}, \exists\text{Needed}.\) 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 \(\Sigma\) 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 DISPOSE1 concerns a situation where below = \(\{\} \land \text{above} \neq \{\}\)\(^2\) — for brevity, this is referred to as “L99”:

\[
\begin{align*}
\text{L99} & \quad \text{inv-Free}(f); \text{disj(locs-of}(d, s), \text{all-locs}(f)); d + s \in \text{dom} f \\
\text{disj(locs-of}(d, s + f(d + s)), \text{all-locs}(\{d + s\} - f))
\end{align*}
\]

\(^2\)The lemma here is one of the four subgoals within \(z\text{-F1.inv\_dispose1\_Disjoint}\) (l. 757 of HEAP1Proofs.thy). This is part of the DISPOSE1 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 = \(\{\} \land \text{above} \neq \{\}\).
3.2. PROOF OBJECTS

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):

- $m: \text{Loc} \rightarrow \mathbb{N}_1$
- $s: \text{Loc-set}$
  
  \[
  \text{all-locs}(s \triangleleft m) \subseteq \text{all-locs}(m)
  \]

- $m: \text{Loc} \rightarrow \mathbb{N}_1$
- $s \in \text{dom } m$
  
  \[
  \forall l, l' \in \text{dom } m \cdot l \neq l' \Rightarrow \text{disj}(\text{locs-of}(l, m(l)), \text{locs-of}(l', m(l')))
  \]
  
  \[
  \text{all-locs}(s \triangleleft m) = \text{all-locs}(m) - \text{locs-of}(s, m(s))
  \]

- $d: \text{Loc}$
- $n, m: \mathbb{N}_1$
  
  \[
  \text{locs-of}(d, n + m) = \text{locs-of}(d, n) \cup \text{locs-of}(d + n, m)
  \]

These three lemmas are stored as Judgements. Here again, establishing apposite lemmas 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 L99) is:

\[
\begin{align*}
\text{from inv-Free1}(f); \text{disj}(\text{locs-of}(d, s), \text{all-locs}(f)); d + s & \in \text{dom } f \\
\text{infer } \text{disj}(\text{locs-of}(d, s + f(d + s)), \text{all-locs}(\{d + s\} \triangleleft f)) & \quad ??
\end{align*}
\]

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 $\text{locs-of}(d, s + f(d + s))$ in the hypothesis. This effectively completes the proof of L99 in that we can fill in its justification — but it has spawned two new unproven conjectures S1 and S3 (the numbering of steps S1 is again indicative of the order of creation); it so happens that S1 has a side condition but this is immediately discharged leading to the following state of proof:

---

3The proof of L3.5 would use inv-Free1 to get:

\[
\forall l, l' \in \text{dom } f \cdot l \neq l' \Rightarrow \text{disj}(\text{locs-of}(l, f(l)), \text{locs-of}(l', f(l')))
\]

which specialises to:

\[
\forall l \in \text{dom } f \cdot l \neq d + s \Rightarrow \text{disj}(\text{locs-of}(d + s, f(d + s)), \text{locs-of}(l, f(l)))
\]

22
CHAPTER 3. MODELS OF WHY

from inv-Free1(f); disj(locs-of(d, s), all-locs(f)); d + s ∈ dom f
S2  \( f(d + s) ∈ \mathbb{N}_1 \)  h[4], Free1
S1  \( \text{locs-of}(d, s + f(d + s)) = \)
\( \text{locs-of}(d, s) \cup \text{locs-of}(d + s, f(d + s)) \)  L4(S2)
S3  disj((locs-of(d, s) \cup \text{locs-of}(d + s, f(d + s))),
\[ \text{all-locs}(\{d + s\} ⊨ f) \])
\[ \text{inferred disj}(\text{locs-of}(d, s + f(d + s)), \text{all-locs}(\{d + s\} ⊨ f)) \]
\[ \text{implies } \text{-subs}(S1, S3) \]

Effectively, the current state σ2 has four conjectures in guts — its domain is \{L99, S3, S1, S2\}.

Turning to the unjustified \( \sigma_2 \) has four conjectures in guts — its domain is \{L99, S3, S1, S2\}.

from inv-Free1(f); disj(locs-of(d, s), all-locs(f)); d + s ∈ dom f
S4  disj(locs-of(d, s), all-locs(\{d + s\} ⊨ f))  ??
S6  disj(locs-of(d + s, f(d + s)), all-locs(\{d + s\} ⊨ f))  ??
S2  \( f(d + s) ∈ \mathbb{N}_1 \)  h[4], Free1
S1  \( \text{locs-of}(d, s + f(d + s)) = \)
\( \text{locs-of}(d, s) \cup \text{locs-of}(d + s, f(d + s)) \)  L4(S2)
S3  disj((locs-of(d, s) \cup \text{locs-of}(d + s, f(d + s))),
\[ \text{all-locs}(\{d + s\} ⊨ f) \])  L2(S4, S6)
\[ \text{inferred disj}(\text{locs-of}(d, s + f(d + s)), \text{all-locs}(\{d + s\} ⊨ f)) \]
\[ \text{implies } \text{-subs}(S1, S3) \]

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 inv-Free1(f); disj(locs-of(d, s), all-locs(f)); d + s ∈ dom f
S5  \[ \text{all-locs}(\{d + s\} ⊨ f) \subseteq \text{all-locs}(f) \]  L3, Free1
S4  disj(locs-of(d, s), all-locs(\{d + s\} ⊨ f))  L1(S5, h[2])
S7  \[ \text{all-locs}(\{d + s\} ⊨ f) = \]
\[ \text{all-locs}(f) − \text{locs-of}(d + s, f(d + s)) \]  L3.5, h[1], h[4]
S6  \[ \text{disj}(\text{locs-of}(d + s, f(d + s)), \text{all-locs}(\{d + s\} ⊨ f)) \]  L1.5, S7
S2  \( f(d + s) ∈ \mathbb{N}_1 \)  h[4], Free1
S1  \( \text{locs-of}(d, s + f(d + s)) = \)
\( \text{locs-of}(d, s) \cup \text{locs-of}(d + s, f(d + s)) \)  L4(S2)
S3  disj((locs-of(d, s) \cup \text{locs-of}(d + s, f(d + s))),
\[ \text{all-locs}(\{d + s\} ⊨ f) \])  L2(S4, S6)
\[ \text{inferred disj}(\text{locs-of}(d, s + f(d + s)), \text{all-locs}(\{d + s\} ⊨ f)) \]
\[ \text{implies } \text{-subs}(S1, S3) \]

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:

\[ \text{Body} :: \cdots \]

\[ \text{strats} : \text{StrId} \rightarrow \text{Strategy} \]

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 AI4FM— e.g. [GKL13].)

\[ \text{Strategy} :: \text{function} : (\text{ToolIP} | \cdots) \]

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

\[ \text{ToolIP} :: \text{name} : \cdots \]

\[ \text{support} : \text{ConjId-set} \]

\[ \text{other} : \cdots \]

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

\[ \text{Strategy} :: \text{function} : (\cdots | \text{Split}) \]

\[ \text{justif} : \text{ConjId} \]

\[ \cdots \]

\[ \text{Split} = \text{Conjecture} \rightarrow \text{Conjecture-set} \]

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]).

\[ \text{Strategy} :: \cdots \]

\[ \text{intent} : [\text{Why}] \]

\[ \text{rank} : \text{Conjecture} \rightarrow \text{Score} \]

\[ \cdots \]

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, NPEANOINDN and NCOMPLETEINDN 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:

```
NPEANOINDN specialises INDUCTIONPROOF
NCOMPLETEINDN specialises INDUCTIONPROOF
```

So the final field becomes:
```
Strategy :: ···
  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 AI4FM 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 :: ···
  rank : Conjecture → Score
  ···
```

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 :: ···
  match : Features
```

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

Features :: provenance : (Origin | Why)*
mainTps : Bddl-set
mainFns : FnId-set
blocks : ConjId-set
other : ···

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 \( d + f(d + s) \) 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 \( \sigma_2 \) 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 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

\[ ^5 \text{Don’t be fooled by the shape of the symbol in the conclusion of L2 it does work like an “and introduction” rule.} \]
CHAPTER 3. MODELS OF WHY

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, NEW1-feas. Here post-NEW1 has a disjunction on whether \( \lnot f_1(r) \) 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 NEW1-feas, the new hypothesis is neither hard to discover nor prove (it follows from the pre-NEW1).

The proof of DISPOSE1-feas benefits from the same strategy but post-DISPOSE1 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 NEW1 followed by DISPOSE1 is an identity over Free1.

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:

\[
\begin{align*}
& \forall l, l' \in \text{dom } \overrightarrow{f} \cdot \text{sep}(l, l', \overrightarrow{f}) \\
& \exists l \in \text{dom } \overrightarrow{f} \cdot \overrightarrow{f}(l) = s \\
& \exists f \in (\text{Loc} \xrightarrow{m} \mathbb{N}_1), r \in \text{Loc} \cdot \\
& \quad r \in \text{dom } f \\
& \quad \overrightarrow{f}(r) = s \land f = \{r\} \in f \\
& \quad \forall l, l' \in \text{dom } f \cdot \text{sep}(l, l', f)
\end{align*}
\]

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:

\[
\begin{align*}
& f_1 \in (\text{Loc} \xrightarrow{m} \mathbb{N}_1), s \in \mathbb{N}_1 \\
& r \in \text{dom } f_1 \\
& f_1(r) = s \\
& \forall l, l' \in \text{dom } f_1 \cdot \text{sep}(l, l', f_1) \\
& f_2 = \{r\} \in f_1 \\
& \forall l, l' \in \text{dom } f_2 \cdot \text{sep}(l, l', f_2)
\end{align*}
\]

assuming \( \text{sep} \) is defined as the second predicate in the Free1-inv definition.

The second spawned conjecture from the case split would be:
3.3. STRATEGIES

\[ \vec{f} \in (\text{Loc} \rightarrow m N_1), s \in N_1 \]
\[ \forall l, l' \in \text{dom} f \cdot \text{sep}(l, l', \vec{f}) \]
\[ \exists l \in \text{dom} f \cdot \vec{f}(l) > s \]
\[ \exists f \in (\text{Loc} \rightarrow m N_1), r \in \text{Loc} \cdot \]
\[ r \in \text{dom} f \wedge \]
\[ \vec{f}(r) > s \land f = \vec{f} \upharpoonright \{r \mapsto \vec{f}(r) - s\} \land \]
\[ \forall l, l' \in \text{dom} f \cdot \text{sep}(l, l', f) \]
\[ f_2 = f_1 \upharpoonright \{r \mapsto f_1(r) - s\} \]
\[ \forall l, l' \in \text{dom} f_2 \cdot \text{sep}(l, l', f_2) \]

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:

\[ f_1 \in (\text{Loc} \rightarrow m N_1), s \in N_1 \]
\[ r \in \text{dom} f_1 \]
\[ f_1(r) > s \]
\[ \forall l, l' \in \text{dom} f_1 \cdot \text{sep}(l, l', f_1) \]
\[ f_2 = f_1 \upharpoonright \{r \mapsto f_1(r) - s\} \]
\[ \forall l, l' \in \text{dom} f_2 \cdot \text{sep}(l, l', f_2) \]

This lemma involves arithmetic and we are less sure what TP systems will make of it. The PO for DISPOSE1 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
CHAPTER 3. MODELS OF WHY

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 \( \text{bdrels} \).

3.3.5 An analogy (formally known as “Leo’s 2c worth”)

Perhaps a useful analogy to interpret \( \text{Conjecture} \) 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 \textit{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 \textit{true} and \textit{thm}. In MWhy, goals (and hypothesis) are \textit{Judgments} representing the term language, with extra (meta-level and structural) information added, whereas \textit{Justification} are brought as a series of (proof) attempts through proof scripts (\textit{Attempt}) or external tools (\textit{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 \textit{Justification} brings to the surface the user intent of a proof attempt, the \textit{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:

\[
\begin{align*}
\Sigma &:: \quad \text{bdm} : \text{BdId} \xrightarrow{m} \text{Body} \\
\quad \text{bdrels} : \text{BdId} \xrightarrow{m} (\text{BdId} \times \text{Relationship})\text{-set}
\end{align*}
\]

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:

\[
\text{Relationship} = \text{Specialisation} | \text{Morphism} | \text{Isomorphism} | \text{Inherits} | \text{Sub} | \text{Similarity} | \cdots
\]

We might, for example, have some very abstract items in \( \text{Body} \) 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.

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

\textit{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

\[ \Sigma ::= \begin{align*}
bdm &: BdId \rightarrow Body \\
bdrels &: BdId \rightarrow (BdId \times Relationship)\text{-}set \\
\end{align*} \]

\[ Body :: uses : BdId\text{-}set \]

\[ \text{domain} : \{\text{RAIL, AUTO,} \ldots\} \]

\[ \text{functions} : FnId \rightarrow FnDefn \]

\[ guts : ConjId \rightarrow Conjecture \]

\[ \text{strats} : StrId \rightarrow Strategy \]

\[ FnDefn :: \begin{align*}
\text{type} &: \text{Signature} \\
\text{defn} &: [\text{Definition}] \end{align*} \]

\[ Conjecture :: \begin{align*}
\text{hyps} &: \text{Judgement}^* \\
\text{goal} &: \text{Judgement} \\
\text{status} &: \{\text{LEMA, REWRITE L2R, NEGATIVE PROPERTY,} \ldots\} \\
\text{justifs} &: \text{JusId} \rightarrow (\text{AXIOM} \mid \text{TRUSTED} \mid \text{Justification}) \\
\text{match} &: \text{Features} \end{align*} \]

\[ \text{Judgement} = \text{Typing} \mid \text{Sequent} \mid \text{Equation} \mid \text{Ordering} \mid \cdots \]

\[ Justification :: \begin{align*}
\text{claim} &: (\text{ConjId} \mid \text{ToolOP}) \\
\text{subst} &: \text{Term} \rightarrow \text{Term} \\
\text{sub-probs} &: \text{ConjId}\text{-}set \end{align*} \]

\[ \text{ToolOP} = \cdots \]

\[ Features :: \begin{align*}
\text{provenance} &: (\text{Origin} \mid \text{Why})^* \\
\text{mainTps} &: BdId\text{-}set \\
\text{mainFns} &: FnId\text{-}set \\
\text{blocks} &: \text{ConjId}\text{-}set \\
\text{other} &: \cdots \end{align*} \]

\[ \text{Origin} = \text{Token} \]

\[ \text{Why} = \text{Token} \]

\[ Strategy :: \begin{align*}
\text{function} &: (\text{ToolIP} \mid \text{Split}) \\
\text{justif} &: \text{ConjId} \\
\text{intent} &: [\text{Why}] \\
\text{rank} &: \text{Conjecture} \rightarrow \text{Score} \\
\text{specialises} &: [\text{StrId}] \end{align*} \]

\[ \text{ToolIP} :: \begin{align*}
\text{name} &: \cdots \\
\text{support} &: \text{ConjId}\text{-}set \\
\text{other} &: \cdots \end{align*} \]

\[ \text{Split} = \text{Conjecture} \rightarrow \text{Conjecture}\text{-}set \]

\[ \text{Why} = \text{Token} \]

\[ Relationship = \text{Specialisation} \mid \text{Morphism} \mid \text{Isomorphism} \mid \text{Inherits} \mid \text{Sub} \mid \text{Similarity} \mid \cdots \]
CHAPTER 3. MODELS OF WHY

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.

∀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.

∃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 NPEANOINDN and NCOMPLETEINDN — 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, ∨-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

   \[ \exists s, t \in \mathbb{N}^* \cdot s \sim t \neq t \sim s \]

   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.
3.5. SUMMARY OF “MODEL”

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 strategies 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 \textit{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 (\textit{i.e.} no design decision is documented in \textit{DISPOSE1} 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.

\[ Loc = \mathbb{N} \]

\[ Free0 = Loc - set \]

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 \((free)\).

\[ has\_seq : Loc^* \times \mathbb{N} \times Loc\_set \rightarrow \mathbb{B} \]

\[ has\_seq(s, n, free) \triangleq \]

\[ \text{elems } s \subseteq free \]

\[ \land \text{len } s = n \land is\_sequential(s) \]

\[ is\_sequential : \mathbb{N}^* \rightarrow \mathbb{B} \]

\[ is\_sequential(s) \triangleq \exists i, j \in \mathbb{N} \cdot \text{elems } s = \{i, \ldots, j\} \]

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.

\[ NEW0 \ (\text{req: } \mathbb{N}) \ \text{res: } Loc\_set \]

\[ \text{ext wr } f_0 : Loc\_set \]

\[ \text{pre } \exists s \in Loc^* \cdot has\_seq(s, \text{req}, f_0) \]

\[ \text{post } \exists s \in Loc^* \cdot (has\_seq(s, \text{req}, f_0) \land \]

\[ \text{res = elems } s \land f_0 = f_0 - res) \]

\[ DISPOSE0 \ (\text{ret: } Loc\_set) \]

\[ \text{ext wr } f_0 : Loc\_set \]

\[ \text{pre } \text{ret} \cap f_0 = \{\} \]

\[ \text{post } f_0 = f_0 \cup \text{ret} \]

\[ ^1 \text{In VDM, sequences are indexed from 1; len returns the sequence size (or length); and elems returns the sequence (range) values as a set.} \]
CHAPTER 4. HOW WE GOT TO WHERE WE ARE

To dispose memory, a set of locations is given to be returned to the free state (i.e. it will be updated, \texttt{wr \_f0}), providing it is not already free (see \texttt{pre}). It is assumed that dispose operations will be called only with locations returned by \texttt{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 (\texttt{Loc^*}). This is modelled using auxiliary function \texttt{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 \texttt{NEW0} allows for zero-memory allocation, which does not seem right (\textit{e.g.} why not \texttt{N1} instead of \texttt{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 \texttt{NEW1} is a memory \texttt{Piece}, instead of a memory \texttt{Location}.

The auxiliary function (\texttt{has\_seq}) creates a protracted “jump” between types (\textit{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 \texttt{NEW0} inefficiency for a search of a suitable set of locations. The state (\texttt{Free1}) is defined as a set of \texttt{Pieces} that neither overlap nor abut their corresponding locations, where \texttt{Piece} is a two-field record containing a location (of type \texttt{Loc}) and size (of type \texttt{N}). In VDM, projection functions are defined for record types (\textit{e.g.} \texttt{LOC(p)} returns a \texttt{Loc} given a \texttt{p} \in \texttt{Piece}).

\texttt{Piece :: LOC : Loc SIZE : N}

\texttt{Free1 = Piece-set}

\texttt{inv (ps) \triangleleft \forall p1, p2 \in ps \cdot}
\texttt{(p1 = p2 \lor locs\_of(p1) \cap locs\_of(p2) = \{}\}
\texttt{\land LOC(p1) + SIZE(p1) \neq LOC(p2))}

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

\texttt{NEW1 (req:N) res:Piece}
\texttt{ext wr f1 : Free1}
\texttt{pre \exists p \in f1 \cdot SIZE(p) \geq req}
\texttt{post locs(f1) = locs(\_f1) - locs\_of(res)}
\texttt{\land locs\_of(res) \subseteq locs(\_f1)}
\texttt{\land SIZE(res) = req}

\texttt{DISPOSE1 (ret:Piece)}
\texttt{ext wr f1 : Free1}
\texttt{pre locs\_of(ret) \cap locs(f1) = \{}\}
\texttt{post locs(f1) = locs(\_f1) \cup locs\_of(ret)
This model employs two auxiliary functions that project a set of locations out of \(\text{Free}1\) and \(\text{Piece}\). Moreover, \(\text{locs}\) is used as the refinement retrieve function linking \(\text{Free}0\) (set of locations) to \(\text{Free}1\) (set of pieces).

\[
\text{locs} : \text{Free}1 \rightarrow \text{Loc-set}
\]

\[
\text{locs}(ps) \triangleq \bigcup \{\text{locs}.of(p) \mid p \in ps\}
\]

**Comments on the model.** The non-zero sizes and heterogeneous operation interfaces are issues that justify adjusting the model accordingly (i.e. \(\text{SIZE} \in \mathbb{N}_1\)). Beyond the unhelpful non-linear equations, which introduce confusing case-analysis, the non-abutting property of the state invariant introduces the complication of \(\text{Piece}\) 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 (\(\text{locs}\)). 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 \(\text{locs}(f_1)\) 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 \(\text{Free}0\) and \(\text{Free}1\) revealed problems with the \(\text{Free}1\) 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 (\(\text{locs}\)) 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 \(\text{Free}1\). Given \(\text{locs}.of\) is defined in terms of a range of locations, we prove a weakening lemma\(^2\) stating that from the definition of subrange and \(\text{locs}.of\), goals involving the \(\text{Free}1\) invariant can be rewritten as

\[
\text{SIZE}(p_1) = 0 \lor \text{SIZE}(p_2) = 0 \lor
\text{LOC}(p_1) + \text{SIZE}(p_1) \leq \text{LOC}(p_2) \lor \text{LOC}(p_2) + \text{SIZE}(p_2) \leq \text{LOC}(p_1)
\]

Considering the predicates involved, say in \(\text{NEW}1\), 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

\[
\text{is\_sequential} \triangleq \{ s : \mathbb{N}^* \mid \exists i, j \in \text{Loc} . \text{ran}(s) = i \ldots j \} \\
\text{has\_seq} \triangleq \{ s : \text{Loc}^* ; n : \mathbb{N} ; f : \text{Free}0 \mid \text{is\_sequential}(s) \land \text{ran}(s) \subseteq f \land \text{dom}(s) = 1 \ldots n \}
\]

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

\[
\ldots \Rightarrow \exists f \in \text{Free}1 \ldots \land \text{locs}(f) = \overset{\text{locs}(f)}{\longrightarrow} - \text{locs\_of}(\text{res})! \ldots
\]

Finding such witness for \( f \) is unnecessarily complicated by the interference of \( \text{locs} \). And in any case, why should the definition of \( \text{NEW}1 \) be in terms of \( \text{locs} \) instead of manipulating the set of \( \text{Piece} \) that is \( \text{Free}1 \)? A similar scenario happens for \( \text{DISPOSE}1 \). 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 \( \text{Piece} \)).

### 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 \( \text{Piece} \) as output to \( \text{NEW}0 \) and input to \( \text{DISPOSE}0 \). This dispenses the use of auxiliary functions at level 0 and the use of \( \text{locs} \) 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 \( \text{locs\_of} \) was now defined in terms of \( \text{Piece} \) as

\[
\text{locs\_of} : \text{Piece} \to \text{Loc\_set}
\]

\[
\text{locs\_of}(p) \triangleq \{ \text{LOC}(p), \ldots, \text{LOC}(p) + \text{SIZE}(p)-1 \}
\]

#### Added automation lemma on numeric ranges

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

\[
\text{lemma}\ \text{locsOfPiece}:
\forall p \in \text{Piece} . \text{locs\_of}(p) = \text{LOC}(p) \ldots \text{LOC}(p) + \text{SIZE}(p)-1
\]

#### Interface and postcondition modifications to level 0

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

\[
\text{NEW}0 \ (\text{req} : \mathbb{N}) \ \text{res} : \text{Piece}
\]

\[
\text{ext}\ \text{wr}\ f_0 : \text{Loc\_set}
\]

\[
\text{pre} \ \exists r \in \text{Piece} . \text{req} = \text{SIZE}(r)
\]

\[
\text{post}\ \text{SIZE}(\text{res}) = \text{req} \land f_0 = \overset{f_0}{\longrightarrow} - \text{locs\_of}(\text{res})
\]

\[
\text{DISPOSE}0 \ (\text{ret} : \text{Piece})
\]

\[
\text{ext}\ \text{wr}\ f_0 : \text{Loc\_set}
\]

\[
\text{pre} \ \text{locs\_of}(\text{ret}) \cap f_0 = \{ \}
\]

\[
\text{post}\ f_0 = \overset{f_0}{\longrightarrow} \cup \text{locs\_of}(\text{ret})
\]

The interface to \( \text{NEW}0 \) now returns a \( \text{Piece} \) instead of a \( \text{Loc\_set} \), and its precondition is simpler: no use of auxiliary functions, and instead it depends on finding a suitable size \( \text{Piece} \) (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 \( \text{locs\_of} \) for \( \text{Piece} \). For \( \text{DISPOSE}0 \), the input is now a \( \text{Piece} \) instead of \( \text{Loc\_set} \). The pre/postconditions are adjusted to use \( \text{locs\_of} \) for making a \( \text{Piece} \) into a contiguous set of locations.
4.2. THEOREM PROVING EXPERIENCES

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.

\[
\text{NEW1 (req: } N \text{) res: Piece}
\]
\[
\text{ext wr } f_1 : \text{Free1}
\]
\[
\text{pre } \exists r \in f_1 \cdot \text{SIZE}(r) \geq \text{req}
\]
\[
\text{post } \exists p \in \text{Piece} \cdot p \in f_1 \wedge \text{locs-of}(p) \subseteq \text{locs-of}(\text{res}) \wedge \text{SIZE}(\text{res}) = \text{req}
\]
\[
f_1 = (f_1 - \{p\}) \cup (\text{locs-of}(p) - \text{locs-of}(\text{res}))
\]

The postcondition is clearly different: instead of relying on locs it explicitly removes the new Piece \( p \in f_1 \) to allocated, where the resulting \( \text{res} \in \text{Piece} \) takes just enough out of the piece \( p \) chosen. Given \( \text{locs-of}(\text{res}) \subseteq \text{locs-of}(p) \wedge \text{SIZE}(\text{res}) = \text{req} \), it is true that \( \text{SIZE}(\text{res}) = \text{req} \). The result \( f_1 \) removes the whole chosen piece \( p \) first then adds the remainder amount from \( p \) not used by \( \text{res} \) (i.e. when \( \text{SIZE}(p) > \text{SIZE}(\text{res}) = \text{req} \)). 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: \( p = \text{res} \), hence

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

simplifies to

\[
(f_1 - \{p\})
\]

which is trivially true, providing \( p \) is instantiated with \( r \in f_1 \) from the preconditions. Nevertheless, this part of the proof led to the following simplification rule lemmas being suggested:

lemma \( \text{lFree1UnitDiff} \)
\[
\forall f \in \text{Free1}, p \in \text{Piece} \cdot f - \{p\} \in \text{Free1}
\]

lemma \( \text{lFree1UnitUnion} \)
\[
\forall f \in \text{Free1}, p \in \text{Piece} \cdot \{p\} \cup f \in \text{Free1}
\]

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
CHAPTER 4. HOW WE GOT TO WHERE WE ARE

the modification on \textit{DISPOSE1}. Nevertheless, in the process we also found some other useful lemmas linking the precondition of \textit{DISPOSE1} and the \textit{Free1} invariant, as well as some general properties about \textit{loc\_of} and \textit{locs}.

\textbf{Lemma} \textit{ILocsWithin}
\[
\forall p, q \in \text{Piece} \cdot \\
\text{LOC}(p) \leq \text{LOC}(q) \land \text{LOC}(q) + \text{SIZE}(q) \leq \text{LOC}(p) + \text{SIZE}(p)
\Rightarrow \text{loc\_of}(q) \subseteq \text{loc\_of}(p)
\]

\textbf{Lemma} \textit{ILocsReminder}
\[
\forall \text{rem, res, p} \in \text{Piece} \cdot \\
\text{SIZE} (\text{rem}) = \text{SIZE}(\text{p}) \cdot \text{SIZE}(\text{res}) \land \\
\text{SIZE}(\text{p}) \geq \text{SIZE}(\text{res}) \land \\
\text{LOC} (\text{res}) = \text{LOC}(p) \Rightarrow \text{loc\_of}(\text{rem}) = \text{loc\_of}(p) - \text{loc\_of}(\text{res})
\]

Lemmas \textit{ILocsWithin} and \textit{ILocsReminder} weakens any goal term involving \textit{loc\_of} subset and set difference as a conjunction of non-linear equations. This is useful when dealing with the postconditions of \textit{NEW1} and \textit{DISPOSE1}. It is also useful as it suggests we might need to think about more general lemmas about \textit{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 \textit{Free1} comes the suggestion for having a similar structure for \textit{locs}, given that it operates on a set of \textit{Piece} just like \textit{Free1} at this point. So the next two lemmas enforce the \textit{Free1} invariant through \textit{locs} regarding the two set function symbols involved (\text{\_\_} and \text{\_\_\_}).

\textbf{Lemma} \textit{ILocsUnitDiff}
\[
\forall f \in \text{Free1}, p \in \text{Piece} \cdot \\
\text{p} \in f \Rightarrow \text{locs}(f - \{p\}) = \text{locs}(f) - \text{loc\_of}(p)
\]

\textbf{Lemma} \textit{ILocsUnitUnion}
\[
\forall f \in \text{Free1}, p \in \text{Piece} \cdot \\
\text{loc\_of}(p) \cap \text{locs}(f) = \{\} \Rightarrow \text{locs}(\{p\} \cup f) = \text{locs}(f) \cup \text{loc\_of}(p)
\]

\subsection{4.2.2.2 Lemma shaping and prover technicalities}

A technical \textbf{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 \textit{Free1} in \textit{IFree1UnitUnion} instead of \textit{f} \cup \{p\} \in \textit{Free1}, and we used \textit{locs}(\{p\} \cup f) = \ldots \text{ in IFree1UnitUnion} instead of \textit{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 \textit{f} \cup \{p\} as \{p\} \cup \textit{f} at the earliest opportunity (\textit{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 \textbf{card} \textit{s} = 0 \Leftrightarrow \textit{s} = [] because one does not want to automatically rewrite every occurrence of \textit{s} = [] into \textbf{card} \textit{s} = 0, yet this is an important useful result.

The same considerations are \textbf{true} in Isabelle, if with slight variations in style and level of detail and control.

\textbf{Rethinking \textit{Free1} invariant}

From the lemmas about \textit{loc\_of} above and from the new postconditions for \textit{NEW1} and \textit{DISPOSE1}, 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 \textit{Free1} invariant itself. For \textit{NEW1} the (almost 16) non-linear equations in the proof were okay,
4.2. THEOREM PROVING EXPERIENCES

but for DISPOSE1, this was clearly unmanageable. We need to rethink the invariant of Free1 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 NEW0, and most changes to the invariant were guided to improve the clarity of what was being modelled by the DISPOSE1 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 ≤ errors (instead of +0 and <).

Inventing new concepts

Firstly, let us remember the original invariant:

\[ \text{Free}_1 = \text{Piece}\text{-set} \]

\[ \text{inv} (ps) \triangleq \forall p_1, p_2 \in ps : (p_1 = p_2 \vee \text{locs}_\text{of}(p_1) \cap \text{locs}_\text{of}(p_2) = \{\}) \]

\[ \land \text{LOC}(p_1) + \text{SIZE}(p_1) \neq \text{LOC}(p_2) \]

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

\[ \forall p_1, p_2 \in ps : (p_1 \neq p_2) \Rightarrow \text{locs}_\text{of}(p_1) \cap \text{locs}_\text{of}(p_2) = \{\} \]

\[ \land \text{LOC}(p_1) + \text{SIZE}(p_1) \neq \text{LOC}(p_2) \]

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:

\[
\text{unique} \quad := \{fr : \text{Piece}\text{-set} | \forall p_1, p_2 \in fr : \text{LOC}(p_1) = \text{LOC}(p_2) \Rightarrow p_1 = p_2\}
\]

\[
\text{before} \quad := \{p_1, p_2 \in \text{Piece} | \text{LOC}(p_1) + \text{SIZE}(p_1) < \text{LOC}(p_2)\}
\]

\[
\text{sep} \quad := \{fr : \text{Piece}\text{-set} | \forall p_1, p_2 \in fr : \text{LOC}(p_1) < \text{LOC}(p_2) \Rightarrow p_1 \text{ before } p_2\}
\]

\[
\text{inv-Free}_1 \quad := \{fr : \text{Piece}\text{-set} | \text{sep}(fr) \land \text{unique}(fr)\}
\]

The original invariant has too weak an invariant for uniqueness of \( \text{Piece} \) \( (p_1 = p_2) \). What really matters is that their locations are unique, rather than the whole Piece (i.e. \( \text{mk-Piece}(0, 5) \neq \text{mk-Piece}(0, 3) \), 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 \( \text{Piece} \) coming before another by stating their locations are apart beyond just the SIZE. Next, we generalise this notion to a whole set of \( \text{Piece} \) and call it separateness between all pieces of a set. Finally, the new invariant for \( \text{Free}_1 \) is defined in terms of a set of \( \text{Piece} \) 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
CHAPTER 4. HOW WE GOT TO WHERE WE ARE

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

New concepts for $\text{DISPOSE1}$ postcondition

The definition of $\text{NEW1}$ carried through from Z/EVES v1 with the slight adjustment about $\text{Piece}$ sizes being $\text{N}_1$ (i.e. it carried a disjunction over each case for equal and greater than, as featured in Section 2.2), but $\text{DISPOSE1}$ also needed new concepts to become both clearer and without the need to refer to $\text{locs}$ in the postcondition. Like with $\text{NEW1}$, we needed to declare its behaviour explicitly.

We toyed around with two concepts:

\begin{itemize}
  \item $\text{wellplaced} := \{ p_1, p_2 \in \text{Piece} \mid \text{unique} (\{p_1, p_2\}) \land (p_1 \text{ before } p_2 \lor p_2 \text{ before } p_1)\}$
  \item $\text{fuse} := \{ p_1, p_2 \in \text{Piece} \mid \text{LOC}(p_1) + \text{SIZE}(p_1) = \text{LOC}(p_2)\}$
\end{itemize}

They were useful for proofs of lemmas involving separability, and in the old definition of $\text{DISPOSE1}$ for Z/EVES v1 of the model. $\text{fuse}$ in particular, features in the final definition of $\text{DISPOSE1}$ given in Section 2.2. Furthermore, we also added the notion of abutting pieces explicitly, again as a (VDM) B-valued function represented as a set (in Z).

\begin{itemize}
  \item $\text{abutt} := \{ p_1, p_2 \in \text{Piece} \mid p_1 \text{ fuse } p_2 \lor p_2 \text{ fuse } p_1\}$
\end{itemize}

The new definition of $\text{NEW1}$ and $\text{DISPOSE1}$ are given as follows. Note the interface change that requested sizes cannot be zero. Also for $\text{NEW1}$, instead of leaving the implicit choice in the post condition as

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

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 $\text{NEW1}$, as described below in the next Sections 4.2.4 onwards.

\begin{itemize}
  \item $\text{NEW1}$ (req:$\text{N}_1$) res: \text{Piece}
  \item $\text{ext wr } f_1 : \text{Free1}$
  \item $\text{pre } \exists r \in f_1 \cdot \text{SIZE}(r) \geq \text{req}$
  \item $\text{post } \exists p \in \text{Piece} \cdot p \in \overrightarrow{f_1} \land \text{SIZE}(p) \geq \text{req} \land \text{res} = \text{mk-Piece}(\text{LOC}(p), \text{req}) \land \text{SIZE}(p) = \text{req} \Rightarrow f_1 = (\overrightarrow{f_1} - \{p\}) \land \text{SIZE}(p) \geq \text{req} \Rightarrow f_1 = (\overrightarrow{f_1} - \{p\}) \cup \{\text{mk-Piece}(\text{LOC}(p) + \text{req}, \text{SIZE}(p)-\text{req})\}$
\end{itemize}

\begin{itemize}
  \item $\text{DISPOSE1}$ (ret: \text{Piece})
  \item $\text{ext wr } f_1 : \text{Free1}$
  \item $\text{pre } \text{locs-of}(\text{ret}) \land \text{locs}(f_1) = \{\}$
  \item $\text{post } \exists \text{join, abut} \in \text{Piece-set} \cdot \text{abt} = \{q \in \text{Piece} \mid q \in f_1 \land \text{ret abutt } q\} \land \text{join} = \{\text{ret}\} \cup \text{abt} \land f_1 = (\overrightarrow{f_1} - \text{join}) \cup \{\text{mk-Piece}(\text{min-loc}(\text{join}), \text{sum-size}(\text{join}))\}$
\end{itemize}
4.2. **THEOREM PROVING EXPERIENCES**

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 ($q \in \mathfrak{f}_1$) that abuts.

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**
  \[
  \forall p, q \in Piece \cdot \\
  p \text{ before } q \Rightarrow \neg q \text{ before } p
  \]

- **Lemma LFree1UniqueUnion**
  \[
  \forall f \in Free^1, p \in Piece \cdot \\
  \text{unique } (f \cup \{p\}) \\
  \Leftrightarrow \\
  (\forall q \in f \cdot LOC(q) = LOC(p) \Rightarrow SIZE(q) = SIZE(p))
  \]

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 $\texttt{nat}$ 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.
CHAPTER 4. HOW WE GOT TO WHERE WE ARE

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 \texttt{locs-of} were translated, and we started making extensive use of Isabelle’s \texttt{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 \texttt{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 \texttt{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 trickier 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 \( \lambda \)-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 \( \lambda \)-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 \texttt{Map.thy} library was our choice.
4.3. SUMMARY

At this stage, for the proof of DISPOSE1 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 AI 4FM. 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 development. 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,
5.2. THE ISABELLE PROOF ASSISTANT

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 procedural 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” 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 *: \[x \notin \text{dom } f\]
shows \[x \notin \text{dom } (s 
-\circ f)\]

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

\footnote{The selection process/criteria here is itself interesting and worth further investigation. How does sledgehammer know what to use/filter?}
CHAPTER 5. HEAP IN ISABELLE

by (metis * domIf 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 \( x \notin \text{dom} (s \cdot\cdot\cdot f) \)
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 Wiedijk'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 \texttt{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:

\begin{verbatim}
have disjoint (locs-of (l + s) (the (f l) - s)) (locs (\{l\} -\cdot\cdot\cdot f))
proof -
  have (locs-of (l + s) (the (f l) - s)) \(\subseteq\) locs-of l (the (f l))
  sorry
  moreover have disjoint (locs-of l (the (f l))) (locs (\{l\} -\cdot\cdot\cdot f))
  sorry
  ultimately show ?thesis by auto
qed
\end{verbatim}

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. THE ISABELLE PROOF ASSISTANT

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:

\[ s \prec m \equiv \lambda x. \text{if } x \in s \text{ then } \text{None else } m \ x \]

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

\[ \text{dom} (S \prec f) = \text{dom} f - S \]

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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Number Lemmas</th>
<th>Freitas Total</th>
<th>Whiteside Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain restriction</td>
<td>(&lt;)</td>
<td>15</td>
<td>15</td>
<td>28</td>
</tr>
<tr>
<td>Domain anti-restriction</td>
<td>(-\prec)</td>
<td>23</td>
<td>80</td>
<td>61</td>
</tr>
<tr>
<td>Map override</td>
<td>(\dagger)</td>
<td>22</td>
<td>54</td>
<td>20</td>
</tr>
<tr>
<td>Map union</td>
<td>(\cup m)</td>
<td>24</td>
<td>71</td>
<td>39</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>92</td>
<td>220</td>
<td>148</td>
</tr>
</tbody>
</table>

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:

\[ \{x\} \prec f = [x \mapsto \text{the} (f x)] \]

describes the result of domain restriction on a singleton set (under the assumption \(x \in \text{dom} f\)). 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:

\[ \text{finite} (\text{dom} (f \cup m g)) \]

\(^2\)Approximately.
CHAPTER 5. HEAP IN ISABELLE

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

3. Finally, there is no $N_1$ datatype in Isabelle. To get around this, we define a predicate
$n1$ and extend it to operate on sets and maps (see Section 5.3.2 for the definitions on
sets and maps).

definition

$n1 :: nat \Rightarrow bool$

where

$n1 n \equiv n > 0$

To make $N_1$ 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 $N_1$!

There is an important difference between the finiteness requirement and the $N_1$
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,
$N_1$ is very much a part of the model; this means that we need to keep track of the VDM $N_1$
type by introducing predicates in many places, resulting in a slightly messy specification and
conditional VDM functions, such as:

definition

$locs-of :: Loc \Rightarrow nat \Rightarrow (Loc \rightarrow \text{set})$

where

$locs-of l n \equiv (\text{if } n1 n \text{ then } \{ i. \ i \geq l \land i < (l + n) \} \text{ else undefined})$

which would not be required if we could specify:

definition

$locs-of-n1 :: Loc \Rightarrow n1 \Rightarrow (Loc \rightarrow \text{set})$

where

$locs-of-n1 l n \equiv \{ i. \ i \geq l \land i < (l + n) \}$

Using this definition, we would need to instantiate $n1$ 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 au-
tomation 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 data-
flow 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 permanent 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:

```plaintext
type-synonym Loc' = nat
type-synonym F0' = Loc' set
```

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

```plaintext
definition is-block :: Loc ⇒ nat ⇒ (Loc set) ⇒ bool
where
is-block l n ls ≡ \( \text{nat1} \) n ∧ \( \text{locs-of} \) 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).

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

```plaintext
definition new0-pre :: F0 ⇒ nat ⇒ bool
where
new0-pre f s ≡ (\( \exists \cdot \) l. (is-block l s f))
```

```plaintext
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)
```

```plaintext
definition dispose0-pre :: F0 ⇒ Loc ⇒ nat ⇒ bool
where
dispose0-pre f d s ≡ (\( \exists \cdot \) l. (is-block l s f))
```

```plaintext
definition dispose0-post :: F0 ⇒ Loc ⇒ nat ⇒ F0 ⇒ bool
where
dispose0-post f d s ≡ (\( \exists \cdot \) l. (is-block l s f))
```

```plaintext
definition dispose0-post :: F0 ⇒ Loc ⇒ nat ⇒ F0 ⇒ bool
where
dispose0-post f d s ≡ (\( \exists \cdot \) l. (is-block l s f))
```
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 packaging 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 adequacy proof (which is independent of the individual operations).

In *level0-basic*, we introduce the state \( f_0 \) and an input \( s_0 \), 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 \( \text{l0-input-notempty-def} \) 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 \( f_0 \) 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 \( d_0 \) of the block the add back to the heap, as in Chapter 2.

```
locale level0-dispose = level0-basic +
```

\(^5\)See http://www.overturetool.org
5.3. THE MODELS IN ISABELLE

```
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 locally 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 Auxiliary 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
```
CHAPTER 5. HEAP IN ISABELLE

\[ \text{nat1-map} :: F1 \Rightarrow \text{bool} \]
where
\[ \text{nat1-map } f \equiv (\forall \cdot x \in \text{dom } f \rightarrow \text{nat1 } (\text{the } (f x))) \]

definition
\[ \text{nat1-set} :: (\text{nat set}) \Rightarrow \text{bool} \]
where
\[ \text{nat1-set } S \equiv (\forall \cdot x \in S \rightarrow \text{nat1 } x) \]

The level 1 model introduces a new auxiliary function, \text{locs} that returns the set of all free locations within a given map. We define the \text{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 \text{nat1-map}:

definition
\[ \text{locs} :: (\text{Loc} \rightarrow \text{nat}) \Rightarrow \text{Loc set} \]
where
\[ \text{locs } sm \equiv (\text{if } \text{nat1-map } sm \text{ then } \bigcup s \in \text{dom } sm. \text{locs-of } s \text{ (the } (sm s)) \text{ else undefined}) \]

It is otherwise \text{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 \text{locs} by giving it a bottom element (as Isabelle’s \text{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 \text{undefined} = \text{undefined} in Isabelle, which is not true in VDM’s three-valued logic. Thus, \text{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:
\[
\begin{align*}
\text{Free}1 &= \text{Loc} \rightarrow \text{m} \rightarrow \mathbb{N}_1 \\
\text{inv } (f) &\triangleq \\
& \forall l, l' \in \text{dom } f. \\
& l \neq l' \Rightarrow \text{is-disj(loocs-of}(l, f(l)), \text{loocs-of}(l', f(l'))) \\
& \forall l \in \text{dom } f \cdot (l + f(l)) \notin \text{dom } f
\end{align*}
\]

It contains two components (a conjunction):

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

We encode these as individual definitions in Isabelle:

definition
\[ \text{Disjoint} :: F1 \Rightarrow \text{bool} \]
where
\[ \text{Disjoint } f \equiv (\forall \cdot a \in \text{dom } f. \forall \cdot b \in \text{dom } f. a \neq b \rightarrow \text{disjoint } (\text{Locs-of } f a) (\text{Locs-of } f b)) \]

definition
\[ \text{sep} :: F1 \Rightarrow \text{bool} \]

53
5.3. THE MODELS IN ISABELLE

where

\( \text{sep } f \equiv (\forall \cdot l \in \text{dom } f . \ l + \text{the}(f l) \notin \text{dom } f) \)

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 \( \text{N}_1 \).

- \( \text{nat1-map} \): that the state doesn’t contain any locations that map to size 0.

- \( \text{finite domain} \): that the domain of the map is finite, similarly to level 0 state.

Thus, the invariant definition is as follows:

definition
\( F1\text{-inv} :: \text{F1} \Rightarrow \text{bool} \)
where \( F1\text{-inv } f \equiv \text{Disjoint } f \land \text{sep } f \land \text{nat1-map } f \land \text{finite}(\text{dom } f) \)

definition
\( \text{VDM-F1-inv} :: \text{F1} \Rightarrow \text{bool} \)
where \( \text{VDM-F1-inv } f \equiv \text{Disjoint } f \land \text{sep } f \)

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 \( \text{invF1E[elim!]} \): \( F1\text{-inv } f \Rightarrow (\text{sep } f \Rightarrow \text{Disjoint } f \Rightarrow \text{nat1-map } f \Rightarrow \text{finite}(\text{dom } f) \Rightarrow R) \Rightarrow R \)
unfolding \( F1\text{-inv-def} \text{ by simp} \)

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 \( \text{sep} \) version here):

lemma \( \text{invF1I[intro]} \): \( F1\text{-inv } f \Rightarrow (\text{sep } f \Rightarrow \text{Disjoint } f \Rightarrow \text{nat1-map } f \Rightarrow \text{finite}(\text{dom } f) \Rightarrow R) \Rightarrow R \)
unfolding \( F1\text{-inv-def} \text{ by simp} \)

lemma \( \text{invF1-sep-weaken} \): \( F1\text{-inv } f \Rightarrow \text{sep } f \)
unfolding \( F1\text{-inv-def} \text{ by simp} \)

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.
CHAPTER 5. HEAP IN ISABELLE

definition new1-pre :: \( F_1 \Rightarrow \text{nat} \Rightarrow \text{bool} \)
where
new1-pre \( f \ s \equiv (\exists \ l \ \in \ \text{dom} \ f . \ \text{the} (f \ l) \geq s) \)
definition new1-post-eq :: \( F_1 \Rightarrow \text{nat} \Rightarrow F_1 \Rightarrow \text{Loc} \Rightarrow \text{bool} \)
where
new1-post-eq \( f \ s \ f' \ r \equiv r \ \in \ \text{dom} \ f \ \land \ \text{the} (f \ r) = s \ \land \ f' = \{ r \} - \triangle f \)
definition new1-post-gr :: \( F_1 \Rightarrow \text{nat} \Rightarrow F_1 \Rightarrow \text{Loc} \Rightarrow \text{bool} \)
where
new1-post-gr \( f \ s \ f' \ r \equiv r \ \in \ \text{dom} \ f \ \land \ \text{the} (f \ r) > s \ \land \ f' = (\{ r \} - \triangle f) \cup m \ [ r + s \mapsto \text{the} (f \ r) - s] \)
definition new1-post :: \( F_1 \Rightarrow \text{nat} \Rightarrow F_1 \Rightarrow \text{Loc} \Rightarrow \text{bool} \)
where
new1-post \( f \ s \ f' \ r \equiv \text{new1-post-eq} \ f \ s \ f' \ r \lor \text{new1-post-gr} \ f \ s \ f' \ r \)

5.3.2.4 DISPOSE operation

Before showing the locale definitions corresponding to the DISPOSE1 operation, we create auxiliary definitions for dispose. The way these came about is discussed in Section 4.2. First are the two auxiliary functions called \( \text{sum-size} \) and \( \text{min-loc} \) which are used in the postcondition are defined using Isabelle’s operators for set minimal and summation, respectively.

definition min-loc :: \( \text{Loc} \Rightarrow \text{nat} \Rightarrow \text{nat} \)
where
min-loc \( sm \) = (if \( sm \neq \text{empty} \) then \( \text{Min} (\text{dom} \ sm) \) else undefined)
definition sum-size :: \( \text{Loc} \Rightarrow \text{nat} \Rightarrow \text{nat} \)
where
sum-size \( sm \) = (if \( sm \neq \text{empty} \) then \( \sum \ x \in (\text{dom} \ sm) . \ \text{the} (sm \ x)) \) else undefined)

Once again, we used Isabelle’s \texttt{undefined} to enable a total function over a subtype, as we did for \( \text{locs} \).

We have two versions of the postconditions: the exact translation from the VDM specification and a version where \texttt{above}, \texttt{below}, and \texttt{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 :: \( F_1 \Rightarrow \text{Loc} \Rightarrow \text{nat} \Rightarrow \text{bool} \)
where
dispose1-pre \( f \ d \ s \equiv \text{disjoint} (\text{locs-of} \ d \ s) (\text{locs} \ f) \)
5.3. THE MODELS IN ISABELLE

**definition**
\(\text{dispose1-post} :: F1 \Rightarrow \text{Loc} \Rightarrow \text{nat} \Rightarrow F1 \Rightarrow \text{bool}\)

**where**
\(\text{dispose1-post} f d s f' \equiv (\exists \cdot \text{below} \ 	ext{above} \ 	ext{ext} .
\begin{align*}
\text{below} & = \{ x \in \text{dom } f . x + \text{the}(f x) = d \} \triangle f \\
\text{above} & = \{ x \in \text{dom } f . x = d + s \} \triangle f \\
\text{ext} & = (\text{above} \cup \text{m below}) \cup \text{m } [d \mapsto s] \\
f' & = ((\text{dom below} \cup \text{dom above}) \triangle f) \cup \text{m } ([\text{min-loc}(\text{ext}) \mapsto \text{sum-size}(\text{ext})])
\end{align*}\)

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

**definition**
\(\text{dispose1-below} :: F1 \Rightarrow \text{Loc} \Rightarrow F1\)

**where**
\(\text{dispose1-below} f d \equiv \{ x \in \text{dom } f . x + \text{the}(f x) = d \} \triangle f\)

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**
\(\text{dispose1-above} :: F1 \Rightarrow \text{Loc} \Rightarrow \text{nat} \Rightarrow F1\)

**where**
\(\text{dispose1-above} f d s \equiv \{ x \in \text{dom } f . x = d + s \} \triangle f\)

**definition**
\(\text{dispose1-ext} :: F1 \Rightarrow \text{Loc} \Rightarrow \text{nat} \Rightarrow F1\)

**where**
\(\text{dispose1-ext} f d s \equiv (\text{dispose1-above} f d s \cup \text{m dispose1-below} f d) \cup \text{m } [d \mapsto s]\)

which allows us to write and prove:

**definition**
\(\text{dispose1-post2} :: F1 \Rightarrow \text{Loc} \Rightarrow \text{nat} \Rightarrow F1 \Rightarrow \text{bool}\)

**where**
\(\text{dispose1-post2} f d s f' \equiv (f' = ((\text{dom dispose1-below} f d) \cup \text{dom dispose1-above} f d s) \triangle f) \cup \text{m } ([\text{min-loc}(\text{dispose1-ext} f d s) \mapsto \text{sum-size}(\text{dispose1-ext} f d s)])\)

**lemma** \(\text{dispose1-equiv}:\)
\(\text{dispose1-post} f d s f' = \text{dispose1-post2} f d s f'\)

**unfolding** \(\text{dispose1-post-defs dispose1-post2-defs}\)

**by** \(\text{auto}\)

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** \(\text{level1-basic} =\)

**fixes** \(f1 :: F1\)
CHAPTER 5. HEAP IN ISABELLE

and $s_1 :: \text{nat}$
assumes $l_1\text{-input-notempty-def} : \text{nat1 s1}$
and $l_1\text{-invariant-def} : F_1\text{-inv f1}$

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

locale level1-new = level1-basic +
assumes $l_1\text{-new1-precondition-def} : \text{new1-pre f1 s1}$

locale level1-dispose = level1-basic +
fixes $d_1 :: \text{Loc}$
assumes $l_1\text{-dispose1-precondition-def} : \text{dispose1-pre f1 d1 s1}$

definition (in level1-new) $\text{new1-postcondition :: F1 \Rightarrow \text{nat} \Rightarrow \text{bool}}$
where
\[
\text{new1-postcondition f' r} \equiv \text{new1-post f1 s1 f' r} \land F_1\text{-inv f'}
\]

definition (in level1-dispose) $\text{dispose1-postcondition :: F1 \Rightarrow \text{bool}}$
where
\[
\text{dispose1-postcondition f' \equiv dispose1-post f1 d1 s1 f' \land F_1\text{-inv f'}}
\]

definition (in level1-dispose) $\text{dispose1-postconditionpsg :: F1 \Rightarrow \text{bool}}$
where
\[
\text{dispose1-postconditionpsg f' \equiv dispose1-post2 f1 d1 s1 f' \land F_1\text{-inv f'}}
\]

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

definition (in level1-new) $\text{PO-new1-feasibility :: bool}$
where
\[
\text{PO-new1-feasibility \equiv (}\exists f' \, r'. \text{new1-postcondition f' r')}
\]

definition (in level1-dispose) $\text{PO-dispose1-feasibility :: bool}$
where
\[
\text{PO-dispose1-feasibility \equiv (}\exists f'. \text{dispose1-postcondition f')}
\]

definition (in level1-dispose) $\text{PO-dispose1-feasibilitypsg :: bool}$
where
\[
\text{PO-dispose1-feasibilitypsg \equiv (}\exists f'. \text{dispose1-postconditionpsg f')}
\]

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 compartmentalised 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;
- Adequacy proof for the reification;
- Widen-precondition for both operations;
- Narrow-postcondition for both operations;
- Sanity proofs that state that, for example, \( \text{DISPOSE(NEW)} = \text{Id} \).

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.

\[
\text{value } [0 \mapsto 4, 6 \mapsto 11]
\]

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

\[
\text{definition } F1-\text{ex-inv} :: F1 \Rightarrow \text{bool}
\]
where
\[ F1-\text{ex-inv } f \equiv F1-\text{inv } f \]

\[
\text{lemmas } F1-\text{ex-inv-defs} = F1-\text{ex-inv-def} F1-\text{inv-defs} F1-\text{ex-def}
\]

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 highlght 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 (\(\text{locs } f1' \subset \text{locs } f1\)). Incidentally, \(\text{locs } f1\) 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”.

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 [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]

\[
\text{context level1-new}
\begin{align*}
\text{definition} \\
PO-new1-postcondition-state-changes :: \text{nat} \Rightarrow \text{bool} \\
\text{where} \\
PO-new1-postcondition-state-changes r \equiv (\forall \cdot f1'. \ new1-postcondition f1' r \rightarrow f1' \neq f1)
\end{align*}
\]

\[
\text{definition} \\
PO-new1-postcondition-state-locs-subset :: \text{nat} \Rightarrow \text{bool} \\
\text{where} \\
PO-new1-postcondition-state-locs-subset r \equiv (\forall \cdot f1'. \ new1-postcondition f1' r \rightarrow \text{locs } f1' \subset \text{locs } f1)
\]

\[
\text{definition} \\
PO-new1-postcondition-diff-f-locs :: \text{nat} \Rightarrow \text{bool} \\
\text{where}
\]

---

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.
5.4. PROOF OF SOME PROPERTIES OF INTEREST

\[ \text{PO-new1-postcondition-diff-f-locs } r \equiv (\forall \cdot f1’. \text{ new1-postcondition } f1’ r \rightarrow \text{locs } f1’ \neq \text{locs } f1) \]

**definition**

\[ \text{PO-new1-postcondition-shrinks-f-locs } :: \text{ nat } \Rightarrow \text{ bool} \]

**where**

\[ \text{PO-new1-postcondition-shrinks-f-locs } r \equiv (\forall \cdot f1’. \text{ new1-postcondition } f1’ r \rightarrow \text{locs } f1’ \subset \text{locs } f1) \]

**definition**

\[ \text{PO-new1-postcondition-f-equiv } :: \text{ nat } \Rightarrow \text{ bool} \]

**where**

\[ \text{PO-new1-postcondition-f-equiv } r \equiv (\forall \cdot f1’. \text{ new1-postcondition } f1’ r \land \text{the}(f1, r) = sI \rightarrow \{r\} \rightarrow f1’ = \{r\} \rightarrow f1) \]

**end**

**definition**

\[ \text{PO-new1-dispose1-identity-post } :: F1 \Rightarrow \text{ nat } \Rightarrow \text{ nat } \Rightarrow \text{ bool} \]

**where**

\[ \text{PO-new1-dispose1-identity-post } f n r \equiv (\forall \cdot f’ f’’. \text{ new1-post } f n f’ r \land \text{dispose1-post } f’ r n f’’ \land F1-inv f \land \text{nat1 } n \rightarrow f = f’’) \]

**definition**

\[ \text{PO-new1-dispose1-identity-pre } :: F1 \Rightarrow \text{ nat } \Rightarrow \text{ nat } \Rightarrow \text{ bool} \]

**where**

\[ \text{PO-new1-dispose1-identity-pre } f n r \equiv (\forall \cdot f’. \text{ new1-pre } f r \land \text{new1-post } f n f’ r \land F1-inv f \land \text{nat1 } n \rightarrow \text{dispose1-pre } f r n) \]
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):

\[ \forall f. s. \ F1-inv f \land nat1 s \land (\exists l \in dom f. s \leq the (f l)) \rightarrow \]
\[ (\exists f'. r. r \in dom f \land \]
\[ (the (f r) = s \land f' = \{ r \} -\triangle f) \lor \]
\[ s < the (f r) \land f' = \{ r \} -\triangle f \cup m [r + s \mapsto the (f r) - s] \land \]
\[ F1-inv f') \]

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:

\[ \forall f. s. \ F1-inv f \land nat1 s \land (\exists l \in dom f. s \leq the (f l)) \rightarrow \]
\[ (\exists f'. r. r \in dom f \land the (f r) = s \land f' = \{ r \} -\triangle f \land F1-inv f') \lor \]
\[ (\exists f'. r. r \in dom f \land s < the (f r) \land \]
\[ f' = \{ r \} -\triangle f \cup m [r + s \mapsto the (f r) - s] \land F1-inv f') \]

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
6.2. FEASIBILITY PROOFS

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: \( f' = \{ r \} \ll sq \) and \( f'' = \{ r \} \ll sq \cup m \) \([r + s \mapsto \text{the} \ (f r) - s]\). 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 existentially 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 \( s < \text{the} \ (f r) \) or \( s = \text{the} \ (f r) \) part of the goals. In the assumptions we have only \( s \leq \text{the} \ (f r) \). This suggests a hidden case analysis on the \( \leq \), leading to the revised goal (which is then split into two subgoals using disjunction elimination):

\[
\forall s. \ F1-inv f \land \text{nat1} s \land (\exists l \in \text{dom} f. \text{the} \ (f l) = s \lor s < \text{the} \ (f l)) \implies \\
(\exists f'. r. \ r \in \text{dom} f \land \text{the} \ (f r) = s \land f' = \{ r \} \ll sq \land \text{F1-inv} f') \lor \\
(\exists f'. r. \ r \in \text{dom} f \land \ s < \text{the} \ (f r) \land \\
f' = \{ r \} \ll sq \cup m \ [r + s \mapsto \text{the} \ (f r) - s] \land \text{F1-inv} f')
\]

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:

\((y \leq x) = (x = y \lor y < x)\)

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. \( f' = \{ r \} \ll sq \) allows us to discharge the first two conjuncts of the goal trivially. The third — the invariant — is basically \( \text{F1-inv} \ (\{ r \} \ll sq) \), which unfolds as:

\[\text{Disjoint} \ (\{ r \} \ll sq) \land \\
\text{sep} \ (\{ r \} \ll sq) \land \text{nat1-map} \ (\{ r \} \ll sq) \land \text{finite} \ (\text{dom} \ (\{ r \} \ll sq))\]

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

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

\[\text{Disjoint} f \implies \text{Disjoint} \ (s \ll sq)\]
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:

\[
\forall \cdot a \in \text{dom} \ (\{r\} -\triangleleft f) \\
\forall \cdot b \in \text{dom} \ (\{r\} -\triangleleft f) \\
a \neq b \rightarrow \text{disjoint} \ (\text{Locs-of} \ (\{r\} -\triangleleft f) \ a) \ (\text{Locs-of} \ (\{r\} -\triangleleft f) \ b) \\
(\forall \cdot l \in \text{dom} \ (\{r\} -\triangleleft f). l + \text{the} \ ((\{r\} -\triangleleft f) l) \not\in \text{dom} \ (\{r\} -\triangleleft f)) \\
(\forall \cdot x \in \text{dom} \ (\{r\} -\triangleleft f) \rightarrow \text{nat1} \ (\text{the} \ ((\{r\} -\triangleleft f) x))) \\
\text{finite} \ (\text{dom} \ (\{r\} -\triangleleft f))
\]

In a more complicated situation like the Mondex example, a naive full expansion 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 \( \text{nat1-map} \ (\{r\} -\triangleleft f) \) and \( \text{finite} \ (\text{dom} \ (\{r\} -\triangleleft f)) \) 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

\( \text{nat1-map} f \rightarrow \text{nat1-map} \ (s -\triangleleft f) \)

\( \text{finite} \ (\text{dom} f) \rightarrow \text{finite} \ (\text{dom} \ (s -\triangleleft f)) \)

\( \text{Disjoint} \ f \rightarrow \text{Disjoint} \ (s -\triangleleft f) \)

and \( \text{sep} f \rightarrow \text{sep} \ (s -\triangleleft f) \).

For the map anti-restriction operation, we only require the \( P f \) assumption to show \( P (s -\triangleleft f) \); 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:

\( \text{Disjoint} \ (\{r\} -\triangleleft f \cup m. [r + s \rightarrow \text{the} \ (f r) - s]) \)

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

---

\(^1\) Could also be seen as a poor mans rippling
6.2. FEASIBILITY PROOFS

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:

\[ a \notin \text{dom } f \]

\[ \text{nat1-map } f \]

\[ \text{nat1 } b \]

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

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 \( \text{locs-of } (x \ (\text{the } (f \ x)) \) 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 \( \text{sep} \) 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 A14FM 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\(^2\):

\[ \text{VDM-F1-inv } f \implies P ((\{r\} - \alpha f) \]

or

\[ [[\text{sep } f; \text{Disjoint } f]] \implies P ((\{r\} - \alpha f) \]

which are equivalent, but the unfolding of \( \text{VDM-F1-inv} \) 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 \( \text{F1-inv} \) for example) if we always expect it to be used in a context where the invariant holds.

\(^2\)Isabelle represents chains of assumptions using \([A; B; C] \implies D\) 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:

\[ \forall f \ d \ s. \quad \text{F1-inv } f \land \text{nat1 } s \land \text{disjoint} \ (\text{locs-of } d \ s) \ (\text{locs } f) \rightarrow (\exists f'. f' :=
\begin{align*}
& (\text{dom} \ (\text{dispose1-below } f \ d) \cup \text{dom} \ (\text{dispose1-above } f \ d \ s)) \lhd f \cup m \\
& \ [\text{min-loc} \ (\text{dispose1-ext } f \ d \ s) \mapsto \text{HEAP1.sum-size} \ (\text{dispose1-ext } f \ d \ s)] \land \\
& \ \text{F1-inv } f') \]
\]

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

\[ \text{F1-inv} \ ((\text{dom} \ (\text{dispose1-below } f \ d) \cup \text{dom} \ (\text{dispose1-above } f \ d \ s)) \lhd f \cup m \\
\ [\text{min-loc} \ (\text{dispose1-ext } f \ d \ s) \mapsto \text{HEAP1.sum-size} \ (\text{dispose1-ext } f \ d \ s)]) \]

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 \textit{above} and \textit{below}:

\[ \text{F1-inv} \ ((\text{dom} \ \{x \in \text{dom } f \mid x + \text{the} \ (f x) = d\} \lhd f) \cup \\
\text{dom} \ \{x \in \text{dom } f \mid x = d + s\} \lhd f) \lhd f \cup m \\
\ [\text{min-loc} \ (\text{min-loc} \ \{x \in \text{dom } f \mid x + \text{the} \ (f x) = d\} \lhd f \cup m \ {x \in \text{dom } f \mid x = d + s\} \lhd f \cup m \\
\ [d \mapsto s])]) \]

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 \textit{dispose1-below } f \ d and \textit{dispose1-above } f \ d \ s. Recall the definitions:

\[ \text{dispose1-below } f \ d \equiv \{x \in \text{dom } f \mid x + \text{the} \ (f x) = d\} \lhd f \]
\[ \text{dispose1-above } f \ d \ s \equiv \{x \in \text{dom } f \mid x = d + s\} \lhd f \]

The filtering equalities force \textit{above} and \textit{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 (\textit{e.g.} the anti restriction

\[ (\text{dom} \ (\text{dispose1-below } f \ d) \cup \text{dom} \ (\text{dispose1-above } f \ d \ s)) \lhd f \]

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

We describe the technique for solving the case where \textit{dispose1-below } f \ d = \{\} and
6.2. FEASIBILITY PROOFS

\(\text{dispose1-above } f d s \neq \{\}\)

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

\[\text{dispose1-ext } f d s \equiv \text{dispose1-above } f d s \cup m \text{dispose1-below } f d + \left[ \right. \]

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

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

which is considerably simpler. In order to expose this as the true proof obligation (under the case analysis), a strategy called we call \(\text{shaping}\) (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:

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

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

and

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

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 \(\text{nat1}\) and \(\text{finite}\) parts of the invariant are trivial. The difficulty is with \(\text{sep}\) and \(\text{Disjoint}\).

For example, the side-conditions for

\[\{a \notin dom f; sep f; \forall \cdot \in \text{dom } f. \ l + \text{the } (f l) \notin \text{dom } [a \mapsto b]; a + b \notin \text{dom } f; \text{nat1 } b\]\n
\[\Rightarrow \text{sep } (f \cup m [a \mapsto b])\]

are:

1. \(d1 \notin dom \ (\{d1 + s1\} -\langle f1\})\), which is easy to solve by automation.
2. \(\text{sep } (\{d1 + s1\} -\langle f1\})\) is solved by further application of invariant breakdown using the before-state invariant hypothesis \((F1-inv f)\).
3. \(\text{nat1 } (\text{the } (f1 (d1 + s1)) + s1)\), which is straightforward for automation to solve.
4. \(\forall \cdot \in \text{dom } (\{d1 + s1\} -\langle f1\}). \ l + \text{the } ((\{d1 + s1\} -\langle f1\} l) \notin \text{dom } [d1 \mapsto \text{the } (f1 (d1 + s1)) + s1])\), which requires some work.
5. \(d1 + \text{the } ((f1 (d1 + s1)) + s1) \notin \text{dom } (\{d1 + s1\} -\langle 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 \(\text{sep0}\) 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 location straight after a chunk of memory may have clarified these conditions e.g. after $s f = s + \text{the}(f s)$ would simplify the first tricky condition to:

$$\forall l \in \text{dom}(\{d1 + s1\} \rightarrow f1). \text{after } l (\{d1 + s1\} \rightarrow f1) \notin \text{dom}(d1 \mapsto \text{the}(f1 (d1 + s1)) + s1)$$

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

For the final goal, the sep part of the invariant allows us to conclude that $d1 + s1 + \text{the}(f1 (d1 + s1)) \notin \text{dom } f1$, which implies that $d1 + s1 + \text{the}(f1 (d1 + s1)) \notin \text{dom}(\{d1 + s1\} \rightarrow f1)$ 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 $\text{dispose1-above } f \ d \ s \neq \{\}$ 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:

- Adequacy: 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 $\exists !f1. f0 = \text{retr0 } f1 \land \text{F1-inv } f1$ (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

\[
\begin{align*}
\text{[finite } F; P \{\}]; \\
\forall F F'. \\
\text{[finite } F; \text{finite } F'; F' \neq \{\}; \text{contiguous } F'; \text{non-abut } F F'; P F] \\
\implies P (F \cup F') \\
\implies P F
\end{align*}
\]

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:

$F = \text{retr0 } f1\text{hook}$

---

3Or after $l f1 \neq d1$. 
6.3. LEVEL 0 AND LEVEL 1 REIFICATION

$F1$-inv $f1hook$

contiguous $F'$

non-abut $F F'$

that

$\exists f1. F \cup F' = \text{retr0 } f1 \land F1$-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:

$f1 = f1hook \cup m \left[ \text{Min } F' \mapsto \text{card } F' \right]$

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

$\text{retr0 } f1 = \text{locs } f1$

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

$\text{locs } (f \cup m g) = \text{locs } f \cup \text{locs } g$

therefore we just need to show that:

$F = \text{locs } f1hook$

and

$F' = \text{locs } [\text{Min } F' \mapsto \text{card } F']$

The first is precisely the induction hypothesis. For the second subgoal, we conjecture that $F' = \text{locs-of } (\text{Min } F') (\text{card } F')$, 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, $\text{locs-of } (\text{Min } F') (\text{card } F')$ gives us a contiguous set of length $\text{card } F'$ starting from $\text{Min } F'$.

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

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

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

which just requires the assumption

$x \notin \text{dom } f$

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 $\text{locs-of}$ over the domain of the map:

$\text{locs } (f \cup m [x \mapsto y]) = (\bigcup_{s \in \text{dom } (f \cup m [x \mapsto y])} \text{locs-of } s (\text{the } ((f \cup m [x \mapsto y]) s)))$
CHAPTER 6. HEAP PROOFS IN ISABELLE

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

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

where the second union is simply \( \text{locs f} \) and we are done.

2. contiguous \( F' = \Rightarrow \text{locs-of (Min F') (card F')} = F' \) is solved with the help of two lemmas:
   one showing that \( \text{Min (locs-of m l)} = m \) and the other that \( \text{card (locs-of m l)} = l \). 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 \( \bigcup m \) operator to \( \bigcup \) and show that both sides were equal in:

\[
F \cup F' = \text{locs (f1hook} \cup m \text{ [Min F' } \mapsto \text{ card F'])}
\]

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

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

To solve this goal, we break down the definition and solve each individual invariant part separately. We take \( \text{sep (f1hook} \cup m \text{ [Min F' } \mapsto \text{ card F'])} \) 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:

\[
\text{Min F'} + \text{ card F'} \notin \text{ dom f1hook}
\]

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:

\[
\text{non-abut F F'}
\]

where

\[
\text{non-abut s1 s2 } \equiv \\
\text{disjoint s1 s2 } \land \ (\forall \cdot l1 \in s1. \forall \cdot l2 \in s2. \ l1 + 1 < l2 \lor l2 + 1 < l1)
\]

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

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

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. LEVEL 0 AND LEVEL 1 REIFICATION

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:

\[
\text{PO-l01-new-widen-pre} \equiv \\
\forall f1 s1. \text{F1-inv } f1 \land \text{nat1 } s1 \land \text{new0-pre } (\text{retr0 } f1) s1 \implies \text{new1-pre } f1 s1
\]

which unfolds to saying (under additional preconditions) if \( \exists l. \text{is-block } l s1 \) (\text{retr0 } f1) then \( \exists l \in \text{dom } f1. \text{nat } s1 \implies \text{the } (f1 l) \). 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 \( l \in \text{dom } f \). This was the first attempt at solving this proof and the incorrectness of the proof step showed itself immediately. Rather, the approach is more subtle: one has to maneuver the goal to find the appropriate witness. The proof sketch used by Whiteside is as follows:

\[
\begin{align*}
\text{have } & \text{locs-subset: locs-of } l s1 \subseteq \text{locs } f1 \\
& \text{sorry} — \text{Show that the locations are indeed with the free space} \\
\text{then have } & l \in \text{locs } f1 \\
& \text{sorry} — \text{Specifically, the first element is in it} \\
\text{then have } & l \in \left( \bigcup s \in \text{dom } f1. \text{locs-of } s \left( \text{the } (f1 s) \right) \right) \\
& \text{sorry} — \text{Unfold the definition of locs} \\
\text{then have } & \exists m \in \text{dom } f1. \text{locs-of } l s1 \subseteq \text{locs-of } m \left( \text{the } (f1 m) \right) \\
& \text{sorry} — \text{Show that locs-of } l s1 \text{ must be contained in one other locs-of} \\
\text{then obtain } & m \text{ where mindom: } m \in \text{dom } f1 \land \\
& \text{locssubm: locs-of } l s1 \subseteq \text{locs-of } m \left( \text{the } (f1 m) \right) \\
& \text{sorry} — \text{Then find an arbitrary } m \text{ that contains the locations from } l \\
\text{then have } & \text{mgrs1: } s1 \leq \text{the } (f1 m) \\
& \text{sorry} — \text{Show that } s1 \text{ must be } s1 \leq m
\end{align*}
\]

Note that the two facts \( \text{mindom} \) and \( \text{mgrs1} \) 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:

\[
\exists m \in \text{dom } f1. \text{locs-of } l s1 \subseteq \text{locs-of } m \left( \text{the } (f1 m) \right)
\]

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 (\( l \in \text{dom } f \)). Thus, we really needed to find the domain element (\textit{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

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

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

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

\(^5\)Recall the discussion about proof sketching in Section 5.2.3.1.
which is similar (analogous) 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 \textit{locs}.

\subsection*{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 \textit{NEW1} is states that if \textit{new1}-post \( f1 \; s1 \; f1' \; r \) then

\[ \textit{new0-post} \; (\textit{retr0} \; f1) \; s1 \; (\textit{retr0} \; f1') \; r \]

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

\[ \text{is-block} \; r \; s1 \; (\text{locs} \; f1) \]

and

\[ \text{locs} \; f1' = \text{locs} \; f1 - \text{locs-of} \; r \; s1 \]

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

\[
\begin{align*}
\text{finite} \; (\text{dom} \; f) \; ; & \; \text{nati-map} \; f \; ; \; \text{Disjoint} \; f \; ; \; l \in \text{dom} \; f \\
\implies \text{locs} \; \{(l) -\Delta f\} = \text{locs} \; f - \text{locs-of} \; l \; (\text{the} \; (f \; l))
\end{align*}
\]

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

\[ f1' = \{r\} -\Delta f1 \cup m \; [r + s1 \mapsto \text{the} \; (f1 \; r) - s1] \]

and we rewrite the \text{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 \textit{below} and \textit{above} being the \textit{Map.empty} map. Then, for each of the case we apply the same \text{locs} distribution lemmas as for new, perform algebraic manipulations, and discharge side-conditions. This is one of the shortest proofs in the level 1, but requires the least thought!

That properly discuss the strategies and reusability.

\section*{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 identities 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\footnote{In fact, Freitas has also formalised the heap store in Z. See Appendix G and model evolution discussion in Chapter 4.}. 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 \textit{gaps} that must...
6.4. SUMMARY

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 divergence 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\textsuperscript{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.

\textsuperscript{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 AI4FM 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 AI4FM 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

¹The 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.
7.2. SYSTEM TO CAPTURE PROOF PROCESS

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 \texttt{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 \texttt{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 \texttt{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 \texttt{ProofProcess} framework include recognition of proof re-runs, capturing back-tracking 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 \texttt{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 \textit{Why} of \texttt{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 \url{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 \url{http://czt.sourceforge.net}.
\(^4\)Eclipse platform. \url{http://www.eclipse.org}
\(^5\)Connected Data Objects (CDO) model repository. \url{http://www.eclipse.org/cdo}
\(^6\)\texttt{ProofProcess} framework is available at \url{https://github.com/andriusvelykis/proofprocess}.
\(^7\)\url{http://andrius.velykis.lt/} and \url{https://github.com/andriusvelykis/proofprocess}.
CHAPTER 7. CONCLUSION

We are also in the process of analysing the data with data mining algorithms, akin to the work done in [HK13].
7.2. SYSTEM TO CAPTURE PROOF PROCESS

Acknowledgements

It is a pleasure to acknowledge the fruitful collaboration with our Scottish colleagues in the AI4FM 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 AI4FM 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.

\[
\begin{align*}
\text{satisfiable} & \quad \exists \sigma \in \Sigma, o \in O \cdot \text{post-OP}(\sigma, i, \sigma, o) \\
& \\
\text{pre-OP}(\sigma, a) & \quad \sigma \in \Sigma, i \in I \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{adequacy} & \quad \exists \sigma_r \in \Sigma_r, \sigma_a \in \Sigma_a \cdot \sigma_r = \text{retr}(\sigma_r) \\
& \\
\text{pre-OP}(\sigma, a) & \quad \sigma \in \Sigma, i \in I \\
\text{pre-OP-A}(\text{retr}(\sigma_r), i) & \quad \sigma_r \in \Sigma_r, i \in I \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{widen-pre} & \quad \exists \sigma_r \in \Sigma_r, i \in I \\
\text{pre-OP-R}(\sigma_r, i) & \quad \text{pre-OP-A}(\text{retr}(\sigma_r), i) \\
\end{align*}
\]

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
A.3. SANITY CHECKS

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

\[
\sigma_r, \tilde{\sigma}_r \in \Sigma_r, i \in I, o \in O
\]

\[
\begin{array}{c}
\text{pre-OP-A}(\text{retr}(\tilde{\sigma}_r), i) \\
\text{narrow-post} \\
\text{post-OP-R}(\sigma_r, i, \sigma_r, o) \\
\text{post-OP-A}(\text{retr}(\tilde{\sigma}_r), i, \text{retr}(\sigma_r), o)
\end{array}
\]

**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 \textit{NEW} followed by \textit{DISPOSE} on the same sizes is the
identity memory. It is also desirable to enforce that \textit{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, P94].

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 \cdot invariant after-state \land 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
```
B.1. THE HEAP IN ISABELLE

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:

• Finally, the specification of the VDM operations 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 f
The 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 s l s n 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 ⇒ s3 ⊆ s2 ⇒ disjoint s1 s3
by (metis Int-absorb2 Int-assoc Int-empty-right disjoint-def le-infI1 order-refl)

lemma l2v0: disjoint s1 s2 ⇒ disjoint s2 s3 ⇒ disjoint (s1 ∪ s2) s3
nitpick
oops

lemma l2v1: s2 ≠ {} ⇒ disjoint s1 s2 ⇒ disjoint s2 s3 ⇒ disjoint (s1 ∪ s2) s3
nitpick
oops

lemma l2v2: s2 ⊆ s1 ⇒ disjoint s1 s3 ⇒ disjoint (s1 ∪ s2) s3
oops

lemma l2: disjoint s1 s2 ⇒ disjoint s1 s3 ⇒ disjoint s2 s3 ⇒ disjoint (s1 ∪ s2) s3
by (metis Un-empty-left disjoint-def inf-sup-distrib2)

lemma l2o: disjoint s1 s3 ⇒ disjoint s2 s3 ⇒ disjoint (s1 ∪ s2) s3
apply (metis Int-commute Un-empty-left disjoint-def inf-sup-distrib1)
done

lemma l3-1: nat1-map f ⇒ nat1-map(S -o f)
by (metis Diff-iff f-in-dom-ar-apply-subsume l-dom-dom-ar nat1-map-def)

lemma l3-2: l ∈ dom (S -o f) ⇒ l ∈ dom f
unfolding dom-antirestr-def
by (cases l∈S, auto)

lemma l3-3: l ∈ dom (S -o f) ⇒ the ((S -o f) l) = the (f l)
unfolding dom-antirestr-def
by (cases l∈S, auto)

lemma l3: nat1-map f ⇒ locs(S -o f) ⊆ locs f
apply (rule subsetI)
C.1. PROCEDURAL ‘THAT LEMMA’

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-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!
thereon 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) -TAIL 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) -TAIL 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

82
APPENDIX C. PROOFS OF CLIFF’S “THAT-LEMMA”

— 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) -a 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 ssbst) — 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) -a f]
```

**apply (rule l2) — S3 : L2(S5, S6)**

**defer**

```
thm l1[of locs-of d s
  locs f
  locs (d+s) -a f]
l3[of f {d+s}]
```

**apply (rule l3) — S4 : L3**

**defer**

```
apply (frule l1[of - locs 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) -a 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 ssbst) — infer : subs(S1) ; Nothing about S6
apply (rule l2) — S3 : L2(S5, S6)
defer
apply (rule l1[of - locs f -]
  simp) — S5 : L1(S4.h)
```

— 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 (rule notE)
apply (rule l7[of d+s f -],simp-all)
```

**apply (fold disjoint-def)**

**apply (unfold disjoint-def)**
C.1. PROCEDURAL ‘THAT LEMMA’

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 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} -o 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)
deffer
apply (rule l1[of - locs f -], simp) — S5 : L1(S4, h4)
apply (rule l3, simp) — S4 : L3(h1[3])
deffer
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.iif:
  nat1-map f =⇒ Disjoint f =⇒ r ∈ dom f =⇒ locs({r} -o f) = locs f - locs-of r (the(f r))
apply (rule equalityI)
APPENDIX C. PROOFS OF CLIFF’S “THAT-LEMMA”

apply (rule-tac [1-] subsetI)
apply (frule-tac [1-] l3-1[of - {x}])
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)
defer
apply (elim conjE)
defer
apply (frule f-in-dom-ar-apply-not-elem singleton-iff)
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
  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) -o f))
  unfolding F1-inv-def
apply (elim conjE)
apply (frule l4[of s the(f(d+s)) d]) — S1 : L4(S2)
apply (rule l3,simp,simp) — S2 : L5(h1[3],h3)
apply (erule ssbst) — 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) -o 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
C.2 ISAR ‘THAT LEMMA’

```isar
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
```

```isar
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} -o 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 subst) — 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
```

```isar
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} -o 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 subst) — 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
APPENDIX C. PROOFS OF CLIFF’S “THAT-LEMMA”

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)

87
C.2. **ISAR ‘THAT LEMMA’**

```plaintext
qed
next
show disjoint (locs-of (d + s) (the (f (d + s))))
  (locs (\{d + s\} - a f))
proof (subst L3pt5) — Step 6: Substitution 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 ≡ (λx. 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 ≡ (λx. if x : s then None else m x)

definition
  ran-antirestr :: ('a → 'b) ⇒ 'b set ⇒ ('a → 'b) (infixl ⤺ 105)
where
  m ⤺ s ≡ (λx. 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: \( x \notin S \implies S \subset \text{insert } x S \)
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)

**lemma** l-diff-un-not-equal: \( R \subset T \implies T \subseteq S \implies S - T \cup R \neq S \)
by auto

D.3 Map operators lemmas

**lemma** l-map-non-empty-has-elem-conv:
\( g \neq \text{empty} \iff (\exists \cdot x . x \in \text{dom } g) \)
by (metis domIff)

**lemma** l-map-non-empty-dom-conv:
\( g \neq \text{empty} \iff \text{dom } g \neq \{\} \)
by (metis dom-eq-empty-conv)

**lemma** l-map-non-empty-ran-conv:
\( g \neq \text{empty} \iff \text{ran } g \neq \{\} \)
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: \( \text{dom}(S \circ g) = S \cap \text{dom } g \)
by (metis Int-commute dom-restr-def dom-restrict)

**lemma** l-dom-r-subset: \( (S \circ g) \subseteq_m g \)
by (metis Int-iff dom-restr-def l-dom-r-iff map-le-def restrict-in)

**lemma** l-dom-r-accum: \( S \circ (T \circ g) = (S \cap T) \circ g \)
by (metis Int-commute dom-restr-def restrict-restrict)

**lemma** l-dom-r-nothing: \( \{\} \circ f = \text{empty} \)
by (metis dom-restr-def restrict-map-to-empty)

**lemma** l-dom-r-empty: \( S \circ \text{empty} = \text{empty} \)
by (metis dom-restr-def restrict-map-empty)
lemma \( l\text{-dom-r-nothing-empty} \): \( S = \{\} \implies S \circ f = \text{Map.empty} \)
by (metis \( l\text{-dom-r-nothing} \))

lemma \( f\text{-in-dom-r-apply-elem} \): \( x \in S \implies (S \circ f) x = (f x) \)
by (metis dom-restr-def restrict-in)

lemma \( f\text{-in-dom-r-apply-the-elem} \): \( x \in \text{dom } f \implies x \in S \implies (S \circ f) x = \text{Some}(the(f x)) \)
by (metis \( f\text{-in-dom-r-apply-elem} \) the.simps)

lemma \( l\text{-dom-r-disjoint-weakening} \): \( A \cap B = \{\} \implies \text{dom}(A \circ f) \cap \text{dom}(B \circ f) = \{\} \)
by (metis dom-restr-def dom-restrict inf-bot-right inf-left-commute restrict-restrict)

lemma \( l\text{-dom-r-subseteq} \): \( S \subseteq \text{dom } f \implies \text{dom}(S \circ f) = S \)
unfolding dom-restr-def
by (metis Int-absorb1 dom-restrict)

lemma \( l\text{-dom-r-dom-subseteq} \): \( \text{dom}(S \circ f) \subseteq \text{dom } f \)
unfolding dom-restr-def
by (auto)

lemma \( l\text{-the-dom-r} \): \( x \in \text{dom } f \implies x \in S \implies \text{the}((S \circ f) x) = \text{the}(f x) \)
by (metis f-in-dom-r-apply-elem)

lemma \( f\text{-in-dom-ar-subsume} \): \( l \in \text{dom}(S -\circ f) \implies l \in \text{dom } f \)
unfolding dom-antirestr-def
by (cases \( l \in S \), auto)

lemma \( f\text{-in-dom-ar-notelem} \): \( l \in \text{dom} (\{r\} -\circ f) \implies l \neq r \)
unfolding dom-antirestr-def
D.3. MAP OPERATORS LEMMAS

by auto

**lemma** f-in-dom-ar-the-subsume:
\[ l \in \text{dom} (S \triangleleft f) \Rightarrow ((S \triangleleft f) l) = (f l) \]
**unfolding** dom-antirestr-def
**by** (cases \( l \in S \), auto)

**lemma** f-in-dom-ar-apply-subsume:
\[ l \in \text{dom} (S \triangleleft f) \Rightarrow ((S \triangleleft f) l) = (f l) \]
**unfolding** dom-antirestr-def
**by** (cases \( l \in S \), auto)

**lemma** f-in-dom-ar-apply-not-elem:
\[ l \notin S \Rightarrow (S \triangleleft f) l = f l \]
**by** (metis dom-antirestr-def)

**lemma** f-dom-ar-subset-dom:
\[ \text{dom}(S \triangleleft f) \subseteq \text{dom} f \]
**unfolding** dom-antirestr-def dom-def
**by** auto

**lemma** l-dom-dom-ar:
\[ \text{dom}(S \triangleleft f) = \text{dom} f - S \]
**unfolding** dom-antirestr-def
**by** (smt Collect-cong domIff dom-def set-diff-eq)

**lemma** l-dom-ar-accum:
\[ S \triangleleft (T \triangleleft f) = (S \cup T) \triangleleft f \]
**unfolding** dom-antirestr-def
**by** auto

**lemma** l-dom-ar-nothing:
\[ S \cap \text{dom} f = \{\} \Rightarrow S \triangleleft f = f \]
**unfolding** dom-antirestr-def
**apply** (simp add: fun-eq-iff)
**by** (metis disjoint-iff-not-equal domIff)

**lemma** l-dom-ar-empty-lhs:
\[ \{\} \triangleleft f = f \]
**by** (metis Int-empty-left l-dom-ar-nothing)

**lemma** l-dom-ar-empty-rhs:
\[ S \triangleleft \text{empty} = \text{empty} \]
**by** (metis Int-empty-right dom-empty l-dom-ar-nothing)
lemma l-dom-ar-everything:
  \( \text{dom } f \subseteq S \implies S - f = \text{empty} \)
by (metis domIff dom-antirestr-def in_mono)

lemma l-map-dom-ar-subset: \( S - f \subseteq m \)
by (metis domIff dom-antirestr-def map-le-def)

lemma l-dom-ar-none: \( \{ \} - f = f \)
unfolding dom-antirestr-def
by (simp add: fun-eq-iff)

lemma l-map-dom-ar-neq: \( S \subseteq \text{dom } f \implies S \neq \{ \} \implies S - f \neq f \)
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 *: \( x \notin \text{dom } f \)
  shows \( x \notin \text{dom } (S - f) \)
by (metis * domIff dom-antirestr-def)

lemma l-dom-ar-not-in-dom2: \( x \in F \implies x \notin \text{dom } (F - f) \)
by (metis domIff dom-antirestr-def)

lemma l-dom-ar-notin-dom-or: \( x \notin \text{dom } f \lor x \in S \implies x \notin \text{dom } (S - f) \)
by (metis Diff_iff l-dom-dom-ar)

lemma l-in-dom-ar: \( x \notin F \implies x \in \text{dom } f \implies x \in \text{dom } (F - f) \)
by (metis f-in-dom-ar-apply-not-elem domIff)

lemma l-dom-ar-insert: \( (\text{insert } x F) - f = \{ x \} - f \)
proof
  fix xa
  show \( (\text{insert } x F - f) \ xa = (\{ x \} - f F - f) \ xa \)
  apply (cases xa)
  apply (simp add: dom-antirestr-def)
  apply (cases xa\in F)
  apply (simp add: dom-antirestr-def)
D.3. MAP OPERATORS LEMMAS

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: \( x \in F \implies (\{ x \} -\circ F -\circ f) = (F -\circ f) \)
by (metis l-dom-ar-insert insert-absorb)

lemma l-dom-ar-disjoint-weakening:
\( \text{dom } f \cap Y = \{ \} \implies \text{dom } (X -\circ f) \cap Y = \{ \} \)
by (metis Diff-Int-distrib2 empty-Diff l-dom-dom-ar)

lemma l-dom-ar-singletons-comm:
\( \{ x \} -\circ \{ y \} -\circ f = \{ y \} -\circ \{ x \} -\circ f \)
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-notin-dom-or l-in-dom-ar l-dom-ar-disjoint-weakening

D.3.0.4 Map override weakening lemmas [EXPERT]

lemma l-dagger-apply:
\( (f \dagger g) x = (\text{if } x \in \text{dom } g \then (g x) \else (f x)) \)
unfolding dagger-def
by (metis (full-types) map-add-dom-app-simps(1) map-add-dom-app-simps(3))

lemma l-dagger-dom:
\( \text{dom}(f \dagger g) = \text{dom } f \cup \text{dom } g \)
unfolding dagger-def
by (metis dom-map-add sup-commute)

lemma l-dagger-lhs-absorb:
\( \text{dom } f \subseteq \text{dom } g \implies f \dagger g = g \)
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:
APPENDIX D. VDM MAPS AUXILIARY LIBRARY

\(\text{dom } f \subseteq \text{dom } g \implies f \dagger g = g\)

**apply** (rule ext)

**apply** (simp add: l-dagger-apply)

**apply** (rule impI)

**find-theorems** - \(\emptyset - \implies\) - name:Set

**apply** (drule contra-subsetD)

**unfolding** dom-def

**by** (simp-all)

**lemma** l-dagger-empty-lhs:

\(\emptyset \dagger f = f\)

**by** (metis dagger-def empty-map-add)

**lemma** l-dagger-empty-rhs:

\(f \dagger \emptyset = f\)

**by** (metis dagger-def map-add-empty)

**lemma** dagger-notemptyL: \(f \neq \emptyset \implies f \dagger g \neq \emptyset\) by (metis dagger-def map-add-None)

**lemma** dagger-notemptyR: \(g \neq \emptyset \implies f \dagger g \neq \emptyset\) by (metis dagger-def map-add-None)

**lemma** l-dagger-dom-ar-assoc:

\(S \cap \text{dom } g = \{\}\ \implies (S -\alpha f) \dagger g = S -\alpha (f \dagger g)\)

**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:

\(g \neq \emptyset \implies f \dagger g \neq \emptyset\)

**by** (metis dagger-def map-add-None)

**lemma** in-dagger-domL:

\(x \in \text{dom } f \implies x \in \text{dom}(f \dagger g)\)

**by** (metis dagger-def domIff map-add-None)

**lemma** in-dagger-domR:

\(x \in \text{dom } g \implies x \in \text{dom}(f \dagger g)\)

**by** (metis dagger-def domIff map-add-None)

**lemma** the-dagger-dom-right:

assumes \(x \in \text{dom } g\)

shows \(\text{the } ((f \dagger g) x) = \text{the } (g x)\)

**by** (metis assms dagger-def map-add-dom-app-simps(1))
D.3. MAP OPERATORS LEMMAS

lemma *the-dagger-dom-left*:
assumes \( x \notin \text{dom} \ g \)
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*:
\[x \neq y = \Rightarrow (f \dagger \{y \mapsto z\}) \ x = f \ x\]
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*:
\[x \notin \text{dom} f \Rightarrow \{x\} -\bowtie (f \dagger \{x \mapsto y\}) = f\]
proof
fix \( z \)
assume \( x \notin \text{dom} f \)
show \((\{x\} -\bowtie (f \dagger \{x \mapsto y\})) \ z = f \ z\)
by (metis \( \langle x \notin \text{dom} f \rangle \) domIff dom-antirestr-def fun-upd-other insertI1 l-dagger-apply singleton-iff)
qed

lemma *antirestr-then-dagger*:
\[r \in \text{dom} f \Rightarrow \{r\} -\bowtie f \dagger \{r \mapsto \text{the} (f \ r)\} = f\]
proof
fix \( x \)
assume \( *: r \in \text{dom} f \)
show \((\{r\} -\bowtie f \dagger \{r \mapsto \text{the} (f \ r)\}) \ x = f \ x\)
proof
(subst l-dagger-apply,simp,intro conjI impl)
  assume \( x=r \) then show Some (the (f \ r)) = f \ r \ using \( * \) by auto
next
  assume \( x \neq r \) then show \((\{r\} -\bowtie f) \ x = f \ x\) by (metis f-in-dom-ar-apply-not-elem singleton-iff)
qed
qed

lemma *dagger-notin-right*:
\[x \notin \text{dom} g \Rightarrow (f \dagger g) \ x = f \ x\]
by (metis l-dagger-apply)

lemma *dagger-notin-left*:
\[x \notin \text{dom} f \Rightarrow (f \dagger g) \ x = g \ x\]
by (metis dagger-def map-add-dom-app-simps(2))

lemma *l-dagger-commute*:
\[\text{dom} f \cap \text{dom} g = \{\} \Rightarrow f \dagger g = g \dagger f\]
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

96
D.3.0.5 Map update weakening lemmas [EXPERT]

without the condition nitpick finds counter example

lemma l-inmapupd-dom-iff:
\[
\begin{align*}
  l \neq x & \implies (l \in \text{dom } (f(x \mapsto y))) = (l \in \text{dom } f)
\end{align*}
\]
by (metis (full-types) domIff fun-upd-apply)

lemma l-inmapupd-dom:
\[
\begin{align*}
  l \in \text{dom } f & \implies l \in \text{dom } (f(x \mapsto y))
\end{align*}
\]
by (metis dom-fun-upd insert-iff option.distinct(1))

lemma l-dom-extend:
\[
\begin{align*}
  x \notin \text{dom } f & \implies \text{dom } (f1(x \mapsto y)) = \text{dom } f1 \cup \{x\}
\end{align*}
\]
by simp

lemma l-updatedom-eq:
\[
\begin{align*}
  x = l & \implies \text{the } (((f \mapsto \text{the } (f(x)) - s)) l) = \text{the } (f l) - s
\end{align*}
\]
by auto

lemma l-updatedom-neq:
\[
\begin{align*}
  x \neq l & \implies \text{the } (((f \mapsto \text{the } (f(x)) - s)) l) = \text{the } (f l)
\end{align*}
\]
by auto

— A helper lemma to have map update when domain is updated

lemma l-insertUpdSpec-aux: dom f = insert x F \implies (f0 = (f1(x \mapsto y))) \implies f = f0 \ (x \mapsto \text{the } (f x))

proof auto

assume insert: \text{dom } f = insert x F
then have \text{x \in dom } f \ by simp
then show f = (f1(x \mapsto y)) \text{ using insert}
  unfolding dom-def
  simp
  (rule ext)
  auto
  done

qed

lemma l-the-map-union-right: \text{x \in dom } g \implies \text{dom } f \cap \text{dom } g = \{\} \implies \text{the } ((f \cup m g) x) = \text{the } (g x)
by (metis l-dagger-apply munion-def)

lemma l-the-map-union-left: \text{x \in dom } f \implies \text{dom } f \cap \text{dom } g = \{\} \implies \text{the } ((f \cup m g) x) = \text{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:
\[
\begin{align*}
  x \notin \text{dom } f & \implies \text{dom } f \cap \text{dom } [x \mapsto y] = \{\}
\end{align*}
\]
by (metis Int-empty-left Int-insert-left dom-eq-singleton-conv inf-commute)

lemma l-munion-apply:
\[
\begin{align*}
  \text{dom } f \cap \text{dom } g = \{\} & \implies (f \cup m g) x = (\text{if } x \in \text{dom } g \text{ then } (g x) \text{ else } (f x))
\end{align*}
\]
unfolding munion-def
D.3. MAP OPERATORS LEMMAS

by (simp add: l-dagger-apply)

lemma l-munion-dom:
  \( \text{dom } f \cap \text{dom } g = \{\} \implies \text{dom}(f \cup m g) = \text{dom } f \cup \text{dom } g \)
unfolding munion-def
by (simp add: l-dagger-dom)

lemma b-dagger-munion-aux:
  \( \text{dom}(\text{dom } g - \triangleleft f) \cap \text{dom } g = \{\} \)
apply (simp add: l-dom-dom-ar)
by (metis Diff-disjoint inf-commute)

lemma b-dagger-munion:
  \( (f \dagger g) = (\text{dom } g - \triangleleft f) \cup m g \)

find-theorems (300) - = (\cdot:(\cdot \Rightarrow \cdot)) - \text{name: Predicate} - \text{name: Product} - \text{name: Quick} - \text{name: New} - \text{name: Record} - \text{name: Quotient} - \text{name: Hilbert} - \text{name: Nitpick} - \text{name: Random} - \text{name: Transitive} - \text{name: Sum-Type} - \text{name: DSeq} - \text{name: Datatype} - \text{name: Enum} - \text{name: Big} - \text{name: Code} - \text{name: Divides}
thm fun-eq-iff[of f \dagger g (dom g - \triangleleft f) \cup 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 conjI impI)
apply (metis l-dagger-assoc)
apply (simp-all add: disjoint-iff-not-equal)
apply blast
apply blast
done

lemma l-munion-assoc:
  \( \text{dom } f \cap \text{dom } g = \{\} \implies \text{dom } g \cap \text{dom } h = \{\} \implies (f \cup m g) \cup m h = f \cup m (g \cup m h) \)
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:
  \( \text{dom } f \cap \text{dom } g = \{\} \implies f \cup m g = g \cup m f \)
by (metis b-dagger-munion l-dagger-commute l-dom-ar-nothing munion-def)

lemma l-munion-subsume:
  \( x \in \text{dom } f \implies \text{the}(f x) = y \implies f = (\{x\} - \triangleleft f) \cup m [x \mapsto y] \)
apply (subst fun-eq-iff)
apply (intro allI)
apply (subgoal-tac dom(\{x\} - \triangleleft f) \cap dom [x \mapsto 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 \( g \subseteq m f \) instead?

lemma l-munion-subsumeG:
APPENDIX D. VDM MAPS AUXILIARY LIBRARY

\[ \text{dom } g \subseteq \text{dom } f \implies \forall x \in \text{dom } g . \ f x = g x \implies f = (\text{dom } g \setminus f) \cup m g \]

unfolding munion-def
apply (subgoal-tac dom (dom g -\(f) \cap \text{dom } g = \{\})
apply simp
apply (subst fun-eq-iff)
apply (rule allI)
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 \text{dom } f \implies \text{dom } f \cap \text{dom } g = \{\} \implies (S -\(f) \cup m g = S -\(f \cup m g) \]

unfolding munion-def
apply (subgoal-tac dom (S -\(f) \cap \text{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:
\[ (f \cup m \text{empty}) = f \]

unfolding munion-def
by (metis dom-empty inf-bot-right l-dagger-empty-rhs)

lemma l-munion-empty-lhs:
\[ (\text{empty} \cup m f) = f \]

unfolding munion-def
by (metis dom-empty inf-bot-left l-dagger-empty-lhs)

lemma l-finite-munion:
\[ \text{finite } (\text{dom } f) \implies \text{finite } (\text{dom } g) \implies \text{dom } f \cap \text{dom } g = \{\} \implies \text{finite } (\text{dom } (f \cup m g)) \]
by (metis finite-Un l-munion-dom)

lemma l-munion-singleton-not-empty:
\[ x \notin \text{dom } f \implies f \cup m [x \mapsto y] \neq \text{empty} \]
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:
\[ \text{dom } f \cap \text{dom } g = \{\} \implies (f \cup m g = \text{empty}) \iff (f = \text{empty} \land g = \text{empty}) \]
apply (rule iffI)
apply (simp only: dom-eq-empty-cone[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 \text{dom } f \implies \{x\} -\(f \cup m [x \mapsto y]) = f \]
apply (subst fun-eq-iff)
D.3. MAP OPERATORS LEMMAS

apply (rule allI)
unfolding dom-antirestr-def
by (auto simp: l-munion-apply)

lemma l-munion-upd: \( \text{dom } f \cap \text{dom } [x \mapsto y] = \{ \} \implies f \cup m [x \mapsto y] = f(x \mapsto y) \)
unfolding munion-def
apply simp
by (metis dagger-def map-add-empty map-add-upd)

lemma munion-notemp-dagger: \( \text{dom } f \cap \text{dom } g = \{ \} \implies f \cup m g \neq \emptyset \implies f \neq g \neq \emptyset \)
by (metis munion-def)

lemma dagger-notemp-munion: \( \text{dom } f \cap \text{dom } g = \{ \} \implies f \neq g \implies f \cup m g \neq \emptyset \)
by (metis munion-def)

lemma munion-notempty-left: \( \text{dom } f \cap \text{dom } g = \{ \} \implies f \neq \emptyset \implies f \cup m g \neq \emptyset \)
by (metis dagger-notemp-munion dagger-notemptyL)

lemma munion-notempty-right: \( \text{dom } f \cap \text{dom } g = \{ \} \implies g \neq \emptyset \implies f \cup m g \neq \emptyset \)
by (metis dagger-notemp-munion dagger-notemptyR)

lemma unionm-in-dom-left: \( x \in \text{dom } (f \cup m g) \implies (\text{dom } f \cap \text{dom } g) = \{ \} \implies x \notin \text{dom } g \implies x \in \text{dom } f \)
by (simp add: l-munion-dom)

lemma unionm-in-dom-right: \( x \in \text{dom } (f \cup m g) \implies (\text{dom } f \cap \text{dom } g) = \{ \} \implies x \notin \text{dom } f \implies x \in \text{dom } g \)
by (simp add: l-munion-dom)


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: \( P \text{ Map.empty} \)
  and insert: \( \\forall e r . \text{finite } (\text{dom } f) \implies e \notin \text{dom } f \implies P f \implies P (f(e \mapsto r)) \)
  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
APPENDIX D. VDM MAPS AUXILIARY LIBRARY

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 \notin dom f0 and P f0 by simp-all
then have PFUpd: P (f0(x -> the (f x))) by (rule insert)
with PFUpd show ?case by simp
qed

lemma finiteRan: finite (dom f) \Rightarrow 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) \Rightarrow finite (dom (\S \leftarrow 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) \Rightarrow finite (dom g) \Rightarrow finite (dom (f \dagger g))
  by (metis dagger-def dom-map-add finite-Un)

lemma finite-singleton: finite (dom [a \mapsto b])
  by (metis dom-eq-singleton-conv finite.emptyI finite-insert)

lemma not-in-dom-ar: finite (dom f) \Rightarrow s \cap dom f = {} \Rightarrow dom (s \ominus 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) \Rightarrow s \cap dom f = {} \Rightarrow dom (s \ominus f) = dom f
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 \ominus (T \ominus f) = T \ominus (S \ominus f)
by (metis l-dom-ar-accum sup-commute)

lemma l-dom-ar-same-subsume-quickspec:
  S \ominus (S \ominus f) = S \ominus f
D.3. MAP OPERATORS LEMMAS

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 ⇒ locs 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 ⇒ finite(locs i n)

by (metis finite-atLeastLessThan b-locs-of-as-set-interval)

lemma b-locs-of-non-empty:
  nat1 n ⇒ locs l n ≠ {}

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 ⇒ card(locs 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
E.3. PROOF OF SOME PROPERTIES OF INTEREST

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
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

\textbf{thm} \texttt{card-Diff-subset[of \ locs-of \ r \ n \ f]}

\textbf{lemma} \texttt{new0-post-shrinks-f-exactly:}
\texttt{PO-new0-post-shrinks-f-exactly}
\textbf{unfolding} \texttt{PO-new0-post-shrinks-f-exactly-def new0-post-def is-block-def}
\textbf{apply} \texttt{safe}
\textit{by} \texttt{(simp add: card-Diff-subset b-locs-of-finite b-locs-of-as-set-interval)}

\textbf{context} \texttt{level0-dispose}
\textbf{begin}

\textbf{lemma} \texttt{dispose0-postcondition-extends-f:}
\texttt{PO-dispose0-postcondition-extends-f}
\textbf{unfolding} \texttt{PO-dispose0-postcondition-extends-f-def dispose0-postcondition-def}
\texttt{dispose0-post-def}
\textit{by} \texttt{(metis Un-commute b-locs-of-non-empty}
\texttt{dispose0-pre-def inf-sup-absorb}
\texttt{inf-sup-ord(3) l0-dispose0-precondition-def}
\texttt{l0-input-notempty-def less-le)}

\textbf{lemma}
\texttt{dispose0-postcondition f' \implies f0 \subseteq f'}
\textbf{unfolding} \texttt{dispose0-postcondition-def dispose0-post-def}
\textit{by} \texttt{(metis b-locs-of-non-empty dispose0-pre-def}
\texttt{l0-dispose0-precondition-def inf-absorb2}
\texttt{inf-commute inf-sup-ord(4) l0-input-notempty-def}
\texttt{le-iff-sup less-le sup_left-idem)}

\textbf{thm} \texttt{card-Un-disjoint[of f0 locs-of d0 s0]}

\textbf{lemma} \texttt{dispose0-postcondition-extends-f-exactly:}
\texttt{PO-dispose0-postcondition-extends-f-exactly}
\textbf{unfolding} \texttt{PO-dispose0-postcondition-extends-f-exactly-def dispose0-postcondition-def dispose0-post-def F0-ine-def}
\textit{by} \texttt{(metis Int-commute add-0-iff card-Un-Int card-empty}
\texttt{dispose0-pre-def finite-Un l0-dispose0-precondition-def}
\texttt{l0-input-notempty-def l-locs-of-card)}

\textbf{lemma}
\texttt{dispose0-postcondition f' \implies card f' = card f0 + s0}
\textbf{unfolding} \texttt{dispose0-postcondition-def dispose0-post-def F0-inv-def}
E.4. GENERAL LEMMAS

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 manageable 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 (backward reasoning) “f_” = deduction from asm (forward reasoning) “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 \Rightarrow x \in \text{dom } f \Rightarrow 0 < (\text{the}(f x))
by (metis nat1-def nat1-map-def)

lemma f-nat1-map-extends-map-le:
g \subseteq m f \Rightarrow nat1-map f \Rightarrow nat1-map g
apply (frule map-le-implies-dom-le)
unfolding map-le-def nat1-map-def
apply (intro allI impI)
**APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)**

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 ⇒ nat1-map (S -α 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 ⇒ nat1-map ({r} -α f)
by (metis k-nat1-map-dom-ar)

lemma l-nat1-map-dagger: nat1-map f ⇒ nat1-map g ⇒ 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 ⇒ nat1-map g ⇒ dom f ∩ dom g = {} ⇒ 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 ⇒ 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 ⇒ Locs-of f l = locs-of l (the (f l))
unfolding Locs-of-def
by simp

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)
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 -a f) ⇒ locs-of x (the((S -a 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 -a 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)
Appendix E. Heap Lemmas and Proofs (Leo)

Lemma l-locs-dom-ar-iff:
\[ \text{nat1-map } f \rightarrow \text{Disjoint } f \rightarrow r \in \text{dom } f \rightarrow \text{locs}(\{r\} -\triangleleft f) = \text{locs } f - \text{locs-of } r \text{ (the}(f r) \text{)} \]
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)
def er
apply (elim conjE)
def er
apply (intro conjI)
apply (metis f-in-dom-ar-subsume f-in-dom-ar-the-subsume)
def er
apply (rule tac x=y in bexI)
apply (metis f-in-dom-ar-apply-not-elem singleton-iff)
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:
\[ \text{nat1-map } f \rightarrow \text{Disjoint } f \rightarrow S \subseteq \text{dom } f \rightarrow \text{locs}(S -\triangleleft f) = \text{locs } f - (\bigcup r \in S . \text{locs-of } r \text{ (the}(f r)) \]
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)
def er
apply (elim conjE)
def er
apply (intro conjI)
apply (metis f-in-dom-ar-subsume f-in-dom-ar-the-subsume)
def er
apply (rule tac [1-] bexE)
apply (rule ballI)
apply (cases S = \{\})
apply (simp add: l-dom-ar-none)
apply (metis l-dom-ar-none)
apply l-dom-ar-none

find theorems simp:- ≠ \{\}
apply (simp add: nonempty iff)
apply (elim exE conjE)
by (metis l-dom-ar-flow l-dom-ar-flow)

Lemma l-locs-empty-iff:
\[ \text{locs empty } = \{\} \]
apply (rule equalityI)
apply (rule tac [1-] subsetI)
simp all
apply (subgoal tac nat1-map empty)
apply (simp add: locs-def)
by (rule l-nat1-map-empty)
E.4. GENERAL LEMMAS

**lemma** l-locs-singleton-iff:
\[\text{nat}1\ y \implies \text{locs} [x \mapsto y] = \text{locs-of} x y\]

**unfolding** locs-def locs-of-def nat1-map-def
by simp

**lemma** f-dom-locs-of:
\[\text{nat}1\text{-map } f \implies (x \in \text{dom } f) \implies (x \in \text{locs-of} x (\text{the } (f x)))\]

**unfolding** locs-of-def
by (simp add: f-nat1-map-nat1-elem)

**lemma** f-in-dom-locs:
\[\text{nat}1\text{-map } f \implies x \in \text{dom } f \implies x \in \text{locs } f\]

apply (simp add: k-in-locs-iff)
apply (rule bexI)
by (simp-all add: f-dom-locs-of)

**lemma** l-locs-munion-iff:
\[\text{nat}1\text{-map } f \implies \text{nat}1\text{-map } g \implies \text{dom } f \cap \text{dom } g = \{\} \implies \text{locs}(f \cup m g) = \text{locs } f \cup \text{locs } g\]

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 [1-2] x=y in bexI)
apply (simp-all)
apply (metis disjoint-iff-not-equal)
by (metis (full-types))

**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:
\[\text{nat}1\text{-map } f \implies \text{nat}1\text{-map } g \implies \text{dom } f \cap \text{dom } g = \{\} \implies \text{locs}(f \dagger g) \subseteq \text{locs } f \cup \text{locs } g\]

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 (simp (full-types))
by (metis (full-types))
Lemma l-locs-dagger-iff:
\[
\text{nat1-map } f \Rightarrow \text{nat1-map } g \Rightarrow (\forall x \in \text{dom } f \cap \text{dom } g . \text{the}(f x) \leq \text{the}(g x)) \Rightarrow \text{locs}(f \dagger g) = \text{locs } f \cup \text{locs } g
\]
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:
\[
\text{finite } (\text{dom } f) \Rightarrow \text{finite } (\text{dom } g) \Rightarrow
\]
\[
g \neq \text{empty} \Rightarrow \text{dom } f \cap \text{dom } g = \{} \Rightarrow
\]
\[
\text{min-loc}(f \cup m g) = (\text{if } f = \text{empty} \text{ then } \text{min-loc } g \text{ else } \min (\text{min-loc } f) (\text{min-loc } g))
\]
unfolding min-loc-def munion-def
by (simp add: l-dagger-not-empty l-dagger-dom Min-Un)

Lemma l-min-loc-singleton:
\[
\text{min-loc } [d \mapsto s] = d
\]
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:
\[
\text{finite } (\text{dom } f) \Rightarrow
\]
\[
\text{dom } f \cap \text{dom } [d \mapsto s] = \{} \Rightarrow
\]
\[
\text{min-loc}(f \cup m [d \mapsto s]) = (\text{if } f = \text{empty} \text{ then } d \text{ else } \min (\text{Min } (\text{dom } f)) d)
\]
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:
\[
\text{finite } (\text{dom } f) \Rightarrow \text{finite } (\text{dom } g) \Rightarrow
\]
\[
g \neq \text{empty} \Rightarrow \text{dom } f \cap \text{dom } g = \{} \Rightarrow
\]
\[
\text{sum-size}(f \cup m g) = (\text{if } f = \text{empty} \text{ then } \text{sum-size } g \text{ else } (\text{sum-size } f) + (\text{sum-size } g))
\]
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 \((\sum \cdot \in \cdot \cdot \cdot) = ((\sum \cdot \in \cdot \cdot \cdot) + ((\sum \cdot \in \cdot \cdot \cdot))\)
thm setsum.F-Un-neutral[of dom f dom g (\lambda x . the (if x \in dom g then g x else f x)), simplified]
thm setsum-Un-disjoint[of dom f dom g (\lambda 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)
E.5. GOAL-ORIENTED - INVARIANT UPDATE

lemma \textit{l-sum-size-singleton}:
\begin{align*}
\text{sum-size } [d \mapsto s] &= s
\end{align*}
unfolding \textit{sum-size-def}
by \textit{simp}

lemma \textit{l-sum-size-munion-singleton}:
\begin{align*}
\text{finite } (\text{dom } f) &\implies
\text{dom } f \cap \text{dom } [d \mapsto s] = \{\} \implies
\text{sum-size}(f \cup \text{m } [d \mapsto s]) = (\text{if } f = \text{empty } \text{then } s \text{ else } \text{sum-size } f + s)
\end{align*}
by (\textit{simp add: l-sum-size-munion l-sum-size-singleton})

E.4.0.6 Other (less useful) lemmas [EXPERT]

lemma \textit{l-disjoint-comm}:
\begin{align*}
(\text{disjoint } A B) = (\text{disjoint } B A)
\end{align*}
by (\textit{metis disjoint-def inf-commute})

lemma \textit{f-F1-inv-disjoint}:
\begin{align*}
F1\text{-inv } f &\implies \text{Disjoint } f
\end{align*}
by (\textit{metis F1-inv-def})

lemma \textit{f-F1-inv-nat1-map}:
\begin{align*}
F1\text{-inv } f &\implies \text{nat1-map } f
\end{align*}
by (\textit{metis F1-inv-def})

lemma \textit{f-F1-inv-sep}:
\begin{align*}
F1\text{-inv } f &\implies \text{sep } f
\end{align*}
by (\textit{metis F1-inv-def})

lemma \textit{f-F1-inv-finite}:
\begin{align*}
F1\text{-inv } f &\implies \text{finite } (\text{dom } f)
\end{align*}
by (\textit{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 \textit{l-sep-singleton}: \text{nat1 } y \implies \text{sep } ([x \mapsto y])
unfolding \textit{sep-def}
by \textit{simp}

definition \textit{sep0} :: \textit{F1} \Rightarrow \textit{F1} \Rightarrow \text{bool}
where
\begin{align*}
\text{sep0 } f g &\equiv (\forall \cdot l \in \text{dom } f . \ l + \text{the} (f l) \notin \text{dom } g)
\end{align*}

lemma \textit{sep0 f f = sep f}
unfolding \textit{sep0-def sep-def}
by \textit{simp}

lemma \textit{l-sep-singleton-upd}:
\begin{align*}
\text{nat1-map } f &\implies x \notin \text{dom } f \implies x + y \notin \text{dom } f \implies \text{nat1 } y \implies \text{sep } f \implies
\text{sep0 } f [x \mapsto y] \implies \text{sep } (f \cup \text{m } [x \mapsto y])
\end{align*}
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:
  \( \text{dom } f \cap \text{dom } g = {} \implies \text{sep } f \implies \text{sep } g \implies \text{sep0 } f \implies \text{sep0 } g \implies \text{sep } (f \cup m \ g) \)
unfolding sep-def sep0-def
by (auto simp: l-munion-dom l-munion-apply)

lemma nat1-map f = \( \implies \) x \( / \) \( \in \) \( \text{dom } f \implies \text{nat1 } y \implies \text{Disjoint } f \implies \text{Disjoint (locs-of } x \ y) \text{ (locs } f \text{)} \implies \text{Disjoint (f } \cup m \ [x \mapsto y]) \)
unfolding Disjoint-def
apply (simp add: l-locs-of-Locs-of-iff)
apply (intro ballI implI)
apply (simp add: l-munion-dom l-munion-apply)
apply (intro conjI implI)
apply (simp-all add: l-disjoint-comm)
unfolding disjoint-def
find-theorems locs-of - - \( \cap \) - = {}
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:
  \( \text{nat1-map } f \implies x \notin \text{dom } f \implies \text{nat1 } y \implies \text{Disjoint } f \implies \text{Disjoint (locs-of } x \ y) \text{ (locs } f \text{)} \implies \text{Disjoint (f } \cup m \ [x \mapsto y]) \)
unfolding Disjoint-def
apply (simp add: l-locs-of-Locs-of-iff)
apply (intro ballI implI)
apply (simp add: l-munion-dom l-munion-apply)
apply (intro conjI implI)
apply (simp-all)
unfolding disjoint-def
find-theorems locs -
find-theorems locs-of - - \( \cap \) - = {}
apply (metis k-inter-locs-iff nat1-def)
by (metis inf-commute k-inter-locs-iff nat1-def)

lemma l-disjoint-singleton: \( \text{Disjoint ([x } \mapsto \ y]) \)
unfolding Disjoint-def
by simp

lemma l-disjoint-munion:
  \( \text{nat1-map } f \implies \text{nat1-map } g \implies \text{Disjoint } f \implies \text{Disjoint } g \implies \text{Disjoint } (f \cup m g) \)
unfolding Disjoint-def
apply (intro implI ballI)
apply (simp add: l-locs-of-Locs-of-iff l-munion-apply l-munion-dom)
apply (intro implI conjI)
apply simp-all
E.5. GOAL-ORIENTED - INVARIANT UPDATE

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)+
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 \Rightarrow x \notin dom f \Rightarrow nat1-map f \Rightarrow 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 \Rightarrow x \notin dom f \Rightarrow finite(dom f) \Rightarrow 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 \Rightarrow Disjoint (S -a f)
by (smt Disjoint-def Locs-of-def domIff dom-antirestr-def)

lemma k-sep-dom-ar:
  sep f \Rightarrow sep (S -a 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) \Rightarrow finite (dom (S -a f))
by (metis finite-subset f-in-dom-ar-subsume subsetI)

lemma k-F1-inv-dom-ar:
  F1-inv f \Rightarrow F1-inv (S -a 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 \( f \uparrow g \), 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 \( \text{dom } f \). 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** \( \text{l-disjoint-mapupd-keep-sep} \):

\[
\begin{align*}
\text{nat1-map } f &\implies \text{Disjoint } f \implies r \in \text{dom } f \implies \text{nat1 } s \implies \text{the}(f r) > s \implies (r+s) \notin \text{dom } f \\
\text{unfolding } \text{Disjoint-def} \\
\text{apply } (\text{erule-tac } x = r \text{ in } \text{ballE}) \\
\text{apply } (\text{erule-tac } x = (r+s) \text{ in } \text{ballE}) \\
\text{apply } (\text{erule impE}) \\
\text{apply } (\text{simp-all}) \\
\text{apply } (\text{rule notI}) \\
\text{apply } (\text{simp add: l-locs-of-Locs-of-iff}) \\
\text{unfolding } \text{disjoint-def} \\
\text{by } (\text{smt k-locs-of-arithIff nat1-map-def})
\end{align*}
\]

**Lemma** \( k-new1-gr-dom-ar-dagger-aux2 \):

\[
\begin{align*}
\text{nat1-map } f &\implies \text{Disjoint } f \implies r \in \text{dom } f \implies \text{nat1 } s \implies \text{the}(f r) > s \implies (r+s) \notin \text{dom } (\{r\} -\triangle f) \\
\text{by } (\text{metis f-in-dom-ar-subsume l-disjoint-mapupd-keep-sep})
\end{align*}
\]

**Lemma** \( k-new1-gr-dom-ar-dagger-aux \):

\[
\begin{align*}
\text{nat1-map } f &\implies \text{Disjoint } f \implies r \in \text{dom } f \implies \text{nat1 } s \implies \text{the}(f r) > s \implies \text{dom } (\{r\} -\triangle f) \cap \text{dom } [r + s \mapsto \text{the}(f r) - s] = \{\} \\
\text{apply } (\text{subst disjoint-iff-not-equal}) \\
\text{by } (\text{metis dom-eq-singleton-conv f-in-dom-ar-subsume l-disjoint-mapupd-keep-sep singletonE})
\end{align*}
\]

**Lemma** \( b-new1-gr-upd-within-req-size \):

\[
\begin{align*}
&\begin{align*}
&\text{r } \in \text{dom } f \implies \text{the}(f r) > s \implies \text{nat1-map } f \implies \\
&\text{locs-of } (r+s) \text{ (the}(f r) - s) \subseteq \text{locs-of } r \text{ (the}(f r)) \\
\end{align*} \\
\text{by } (\text{simp add: b-locs-of-as-set-interval})
\end{align*}
\]

**Lemma** \( b-new1-gr-upd-psubset-req-size \):

\[
\begin{align*}
&\begin{align*}
&\text{nat1 } s \implies \text{r } \in \text{dom } f \implies \text{the}(f r) > s \implies \text{nat1-map } f \implies \\
&\text{locs-of } (r+s) \text{ (the}(f r) - s) \subset \text{locs-of } r \text{ (the}(f r)) \\
\end{align*} \\
\text{apply } (\text{rule psubsetI}) \\
\text{apply } (\text{simp add: b-new1-gr-upd-within-req-size}) \\
\text{apply } (\text{simp add: b-locs-of-as-set-interval}) \\
\text{by } (\text{metis add-0-iff add-lessD1 add-less-cancel-left atLeastLessThan-inj(1) not-less0})
\end{align*}
\]
E.5. GOAL-ORIENTED - INVARIANT UPDATE

**Lemma** \(k\)-Disjoint-dom-ar-dagger:
\[
    r \in \text{dom} f \implies (f r) > s \implies \text{nat1-map} f \implies \text{Disjoint} f \implies \text{Disjoint} ((\{r\} \cdot -\alpha f) \uplus [r + s \mapsto \text{the}(f r) - s])
\]

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 (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 add-lessD1)
done

**Lemma** \(k\)-Disjoint-dom-ar-munion:
\[
    r \in \text{dom} f \implies (f r) > s \implies \text{nat1-map} f \implies \text{Disjoint} f \implies \text{Disjoint} ((\{r\} \cdot -\alpha f) \cup m [r + s \mapsto \text{the}(f r) - s])
\]

apply (insert k-sep-dom-ar-dagger-aux2 [of s r f])
apply (simp add: l-dagger-dom-ar-assoc)
unfolding sep-def
apply (intro ballI)
apply (simp add: l-dagger-dom)
apply (intro impI conjI)

**Lemma** \(k\)-sep-dom-ar-dagger-aux2:
\[
    \text{nat1} s \implies \{r\} \cap \text{dom} [r + s \mapsto \text{the}(f r) - s] = \{\}
\]

apply (subt disjoint-iff-not-equal)
by auto

**Lemma** \(k\)-sep-dom-ar-dagger:
\[
    r \in \text{dom} f \implies (f r) > s \implies \text{nat1} s \implies \text{sep} f \implies \text{sep} ((\{r\} \cdot -\alpha f) \uplus [r + s \mapsto \text{the}(f r) - s])
\]

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-dom)
apply (intro impI conjI)
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

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 \Rightarrow r \in \text{dom } f \Rightarrow (f r) > s \Rightarrow nat1 s \Rightarrow sep f \Rightarrow Disjoint f \Rightarrow sep (\{r\} -\triangle f \cup m [r + s \mapsto \text{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 \Rightarrow r \in \text{dom } f \Rightarrow (f r) > s \Rightarrow nat1-map f \Rightarrow nat1-map (\{r\} -\triangle f \uplus m [r + s \mapsto \text{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 \Rightarrow r \in \text{dom } f \Rightarrow (f r) > s \Rightarrow Disjoint f \Rightarrow nat1-map f \Rightarrow nat1-map (\{r\} -\triangle f \cup m [r + s \mapsto \text{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 \text{dom } f \Rightarrow (f r) > s \Rightarrow \text{finite } (\text{dom } f) \Rightarrow \text{finite } (\text{dom}(\{r\} -\triangle f \uplus m [r + s \mapsto \text{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 \text{dom } f \Rightarrow (f r) > s \Rightarrow nat1 s \Rightarrow nat1-map f \Rightarrow Disjoint f \Rightarrow \text{finite } (\text{dom } f) \Rightarrow 
\text{finite } (\text{dom}(\{r\} -\triangle f \cup m [r + s \mapsto \text{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 \text{dom } f \Rightarrow r \in \text{dom } f \Rightarrow (f r) > s \Rightarrow \text{finite } (\text{dom } f) \Rightarrow \text{finite } (\text{dom}(\{r\} -\triangle f \cup m [r f-munion-ALT-PROOF)

by (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 \text{dom } f \Rightarrow r \in \text{dom } f \Rightarrow (f r) > s \Rightarrow \text{finite } (\text{dom } f) \Rightarrow \text{finite } (\text{dom}(\{r\} -\triangle f \cup m [r (\text{dom}(\{r\} -\triangle f \cup m [r + s \mapsto \text{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)
E.6. GOAL-ORIENTED - DISPOSE1 INVARIANT UPDATE

+ s ↦ the (f r - s))
  thm l-munion-dom[of {r} -of [r + s ↦ the(f r) - s]]
apply (insert l-munion-dom[of {r} -of [r + s ↦ 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-inv f ⇒ nat1 s ⇒ r ∈ dom f ⇒ the(f r) > s ⇒ F1-inv({r} -of \( f \cup [r + s ↦ the(f r) - s] \))
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 ▷ g ≠ empty ⇒ min-loc (S ▷ g) = Min (S ∩ dom g)
by (metis min-loc-def l-dom-r-iff)

lemma k-Min-subset:
  S ≠ {} ⇒ finite T ⇒ S ⊆ T ⇒ Min S ∈ T
by (metis Min-in finite-subset set-mp)

lemma k-min-loc-dom:
  g ≠ empty ⇒ finite (dom g) ⇒ dom g ⊆ dom f ⇒ min-loc g ∈ dom f
unfolding min-loc-def
by (metis Min-in dom-eq-empty-conv set-mp)

lemma k-dispose-abovebelow-dom-disjoint:
  nat1 s1 ⇒ dom (dispose1-above f1 d1 s1) ∩ dom (dispose1-below f1 d1) = {}
find-theorems \( \cap \) 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 ⇒ d1 \∉ dom (dispose1-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-map f1 ⇒ nat1 s1 ⇒ d1 \∉ dom (dispose1-below f1 d1)
unfolding dispose1-below-def
find-theorems dom(\- \-)

118
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

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 \Rightarrow sep f1 \Rightarrow nat1 s1 \Rightarrow 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 \Rightarrow dom (dispose1-above f1 d1 s1 \cup m dispose1-below f1 d1)
  = \{ x \in dom f1 . x + the(f1 x) = d1 \lor 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) \Rightarrow 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) \Rightarrow 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) \Rightarrow nat1 s1 \Rightarrow 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 \Rightarrow (dispose1-above f1 d1 s1) = Map.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 \Rightarrow x + the(f1 x) = d1 \Rightarrow (dispose1-below f1 d1) \neq Map.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 \Rightarrow d1 + s1 \in dom f1 \lor x \in dom f1 \land x + the(f1 x) = d1 \Rightarrow
  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:
  natI s1 ⟹ sep0 [d1 ↦→ s1] f1 ⟹ sep0 f1 [d1 ↦→ s1] ⟹
  dispose1-above f1 d1 s1 ∪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 ↦→ s1] f1 ⟹ dispose1-above f1 d1 s1 = empty
apply (simp only: dom-eq-empty-conv[symmetric])
unfolding sep0-def dispose1-above-def
find-theorems dom(- -)
apply (simp add: dom-eq-empty-conv[symmetric] l-dom-r-iff)
by blast

lemma k-dispose1-sep0-below-empty:
  sep0 f1 [d1 ↦→ s1] ⟹ 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 ↦→ 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 (- -)
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 ↦→ 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 (- -)
apply (simp add: f-in-dom-r-apply-elem)
by (metis domIff)

lemma f-dispose1-pre-not-in-dom:
  nat1-map f \implies nat1 s \implies locs-of d s \cap locs f = \{\} \implies 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 \implies dispose1-above f1 d1 s1 = [d1+s1 \mapsto \text{the}(f1 (d1+s1))]
unfolding dispose1-above-def
apply (subst fun-eq-iff)
apply (rule allI)
find-theorems (- :: -)
unfolding dom-restr-def
by auto

lemma l-dispose1-nonempty-above-singleton:
  dispose1-above f1 d1 s1 \neq \text{empty} \implies dispose1-above f1 d1 s1 = [d1+s1 \mapsto \text{the}(f1 (d1+s1))]
by (metis k-empty-dispose-above l-dispose1-above-singleton)

lemma a = x \implies ([x \mapsto y] a) = Some y
by simp

lemma a \neq x \implies ([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
E.6. GOAL-ORIENTED - DISPOSE1 INVARIANT UPDATE

apply auto
done

lemma l ∈ dom fbelow ⇒ l+the(fbelow l)=11 ⇒ dispose1-below fbelow 11 = [l→the(fbelow l)]
unfolding fbelow-def dispose1-below-def
apply safe
apply (simp add: fun-eq-iff)
apply (intro conjI allI impI)
apply (simp-split: split-if-asn)
unfolding dom-restr-def restrict-map-def
apply simp
apply auto
done

unfolding dom-restr-def restrict-map-def
apply simp
apply (erule conjE)
apply (erule_tac x=l in ballE)
find-theorems simp:- = (::('a ⇒ 'b)) -name:HEAP -name:VDM
apply (subst fun-eq-iff)
apply simp
apply (intro allI impI conjI)
apply (simp all: f-in-dom-r-apply-the-elem)
unfolding dom-restr-def restrict-map-def
apply (simp-split: split-if-asn)
unfolding dom-restr-def restrict-map-def
apply (simp-split: split-if-asn)
apply (rule impI)
apply (erule conjE)
apply (erule 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-split: split-if-asn)
by (metis l-dispose1-below-singleton-useless)

lemma l-dispose1-below-singleton-useless:
l ∈ dom f ⇒ l+the(f l) = d ⇒ nat1-map f ⇒ sep f ⇒ Disjoint f ⇒ dispose1-below f d = [l ⇒ 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 all: f-in-dom-r-apply-the-elem)
unfolding dom-restr-def restrict-map-def
apply (simp-split: split-if-asn)
unfolding dom-restr-def restrict-map-def
apply (simp-split: split-if-asn)
apply (rule impI)
apply (erule conjE)
apply (erule 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-split: split-if-asn)
by (metis l-dispose1-below-singleton-useless)

lemma l-sum-size-upd:
finite(dom f) ⇒ x /∈ dom f ⇒ sum-size(f(x→y)) = (if f = empty then y else sum-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 x a . the (if xa = x then Some y else f xa)) (λ x . the (f x))]

lemma l-nat1-sum-size-dispose1-ext:
nat1-map f1 ⇒ finite (dom f1) ⇒ sep f1 ⇒ nat1 s1 ⇒ nat1 (sum-size (dispose1-ext f1 d1 s1))
unfolding dispose1-ext-def
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

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 =⇒ sep f =⇒ Disjoint f =⇒ nat1 s =⇒ d + s ∉ dom (dispose1-below f d)
apply (cases dispose1-below f d = empty)
apply simp
apply (simp add: l-dispose1-below-singleton-useful)
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) =⇒ nat1-map f =⇒ Disjoint f =⇒ sep f =⇒ nat1 s =⇒
  min-loc(disse 1-ext f d s) = min-loc(dispose1-below f d ∪ m[d↦→s])
unfolding dispose1-ext-def
apply (cases dispose1-below f d s = empty)
apply (simp add: l-dispose1-below-singleton-useful)
apply (simp add: l-dispose1-nonempty-above-singleton)
thm l-munion-commute[of [d + s ↦ the (f (d + s))] dispose1-below f d ∪ m[d↦→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 (- ∪m -)
thm k-min-loc-munion-singleton[of dispose1-below f d ∪ m [d ↦ 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-dom-dispose)
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 impl)
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 (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 simp: - ∈ 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)
apply (frule f-nat1-map-nat1-elem,simp-all)
apply (metis sep-def)
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 simp: - ∈ 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)
apply (frule f-nat1-map-nat1-elem,simp-all)
apply (metis sep-def)
apply (metis sep-def)
done
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

lemma l-disjoint-dispose1-ext:
finite(dom f1) ⇒ nat1-map f1 ⇒ Disjoint f1 ⇒ sep f1 ⇒ nat1 s1 ⇒ dispose1-pre f1 d1 s1
E.6. GOAL-ORIENTED - DISPOSE1 INVARIANT UPDATE

⇒

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))) -ο f1)
apply (simp add: l-min-loc-dispose1-ext-absorb-above)

find-theorems simp:(- Ω -)
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)
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)
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 (cases dispose1-below f1 d1 s1 = 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-below 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-below-empty-iff if f1 d1 s1)
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 (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 if 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 Lright_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
E.7. NEW 1 PROOFS

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 names 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 subgoal 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 names 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 names 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 names 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
operator, 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 anticipate that these lemmas will be reused in other VDM-style map prob-
lems. 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 new1_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} -\triangle f1)
by (metis F1-inv-def k-Disjoint-dom-ar l1-invariant-def)

lemma k-new1-sep-dom-ar:
    sep ({r} -\triangle f1)
by (metis F1-inv-def k-sep-dom-ar l1-invariant-def)

lemma k-new1-nat1-map-dom-ar:
    nat1-map ({r} -\triangle f1)
by (metis F1-inv-def k-nat1-map-dom-ar l1-invariant-def)

lemma k-new1-finite-dom-ar:
    finite (dom ({r} -\triangle f1))
by (metis F1-inv-def k-finite-dom-ar l1-invariant-def)
E.7. NEW 1 PROOFS

E.7.1.2 NEW 1 FSB weakening lemmas for greater than case

For new1_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 new1_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 that 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 directly from general lemmas about the involved operators and are all sledgehammered.

**lemma** k-new1-Disjoint-dom-ar-munion:

\[ r \in \text{dom } f1 \Rightarrow \text{the } (f1 r) > s1 \Rightarrow \text{Disjoint } (\{r\} -\langle f1 \cup m [r + s1 \mapsto \text{the } (f1 r) - s1]) \]

**by** (smt F1-inv-def k-Disjoint-dom-ar-munion l1-input-notempty-def l1-invariant-def)

**lemma** k-new1-sep-dom-ar-munion:

\[ r \in \text{dom } f1 \Rightarrow \text{the } (f1 r) > s1 \Rightarrow \text{sep } (\{r\} -\langle f1 \cup m [r + s1 \mapsto \text{the } (f1 r) - s1]) \]

**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:

\[ r \in \text{dom } f1 \Rightarrow \text{the } (f1 r) > s1 \Rightarrow \text{nat1-map } (\{r\} -\langle f1 \cup m [r + s1 \mapsto \text{the } (f1 r) - s1]) \]

**by** (metis F1-inv-def k-nat1-map-dom-ar-munion l1-input-notempty-def l1-invariant-def)

**lemma** k-new1-finite-dom-ar-munion:

\[ r \in \text{dom } f1 \Rightarrow \text{the } (f1 r) > s1 \Rightarrow \text{finite } (\text{dom}(\{r\} -\langle f1 \cup m [r + s1 \mapsto \text{the } (f1 r) - s1])) \]

**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.
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

lemma zw-new1-post-eq:
  \( r \in \text{dom}\ f_1 \implies (f_1 r) = s_1 \implies \text{new1-post-eq}(f_1 s_1 \setminus \{r\}, f_1) \) r

unfolding new1-post-eq-def
by auto

lemma zw-F1-inv-new1-eq:
  \( r \in \text{dom}\ f_1 \implies (f_1 r) = s_1 \implies \text{F1-inv}(\{r\}, f_1) \) r

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:
  \( r \in \text{dom}\ f_1 \implies (f_1 r) > s_1 \implies \text{new1-post-gr}(f_1 s_1 \setminus \{r\}, f_1 \cup m[r + s_1 \mapsto (f_1 r) - s_1]) \) r

unfolding new1-post-gr-def
by auto

lemma zw-F1-inv-new1-gr:
  \( r \in \text{dom}\ f_1 \implies (f_1 r) > s_1 \implies \text{F1-inv}(\{r\}, f_1 \cup m[r + s_1 \mapsto (f_1 r) - s_1]) \) r

by (metis zw-new1-post-eq zw-F1-inv-new1-eq)

E.7.1.4 NEW 1 FSB main theorem

Finally, the top-down strategy applies zooming and weakening patterns, once the key point about splitting exact and surplus memory allocation is observed\(^1\).

theorem locale1-new-FSB: \( \text{PO-new1-feasibility} \)
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 - = \{(\cdot < \cdot) \lor -\}
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_dispose1_munion_disjoint.

context level1-dispose

\(^1\)Oddly enough, we are saying “finally” for where usually is the place work begins!
E.8. DISPOSE 1 PROOFS

begin

lemma k-dispose-above-ext-dom-disjoint-aux:
\[ d1 \notin \text{dom} \ (\text{dispose1-above } f1 \ d1 \ s1) \]
by (metis f-d1-not-dispose-above l1-input-notempty-def)

lemma k-dispose-below-ext-dom-disjoint-aux:
\[ d1 \notin \text{dom} \ (\text{dispose1-below } f1 \ d1) \]
by (metis f-d1-not-dispose-below l1-invariant-def F1-inv-def l1-input-notempty-def)

lemma k-finite-dispose-above-aux:
\[ \text{finite} \ (\text{dom} \ (\text{dispose1-above } f1 \ d1 \ s1)) \]
by (metis f-F1-inv-finite k-finite-dispose-above l1-invariant-def)

lemma k-finite-dispose-below-aux:
\[ \text{finite} \ (\text{dom} \ (\text{dispose1-below } f1 \ d1)) \]
by (metis f-F1-inv-finite k-finite-dispose-below l1-invariant-def)

Now for the \text{*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 \((l\text{min}\_loc\_dispose1\_ext\_iff)\). 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:
\[ \text{dom} \ ((\text{dom} \ (\text{dispose1-below } f1 \ d1)) \cup \text{dom} \ (\text{dispose1-above } f1 \ d1 \ s1)) \setminus f1) \cap \text{dom} \ [\text{min-loc} \ (\text{dispose1-ext } f1 \ d1 \ s1) \mapsto \text{HEAP1}\_\text{sum-size} \ (\text{dispose1-ext } f1 \ d1 \ s1)] = \{\} \]
— simp would do as well

find-theorems - \cap - = \{\}

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)

unfolding dispose1-pre-def disjoint-def
apply (insert l1-dispose1-pre-condition-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 (- \setminus 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 (\lambda x . x \in \text{dom } f1)]
apply (simp add: k-min-loc-munion-singleton[simplified]
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

k-finite-dispose-above-aux
k-dispose-above-ext-dom-disjoint-aux
split: split-if-asn)
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-asn
  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=
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-asn)
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-asn
  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)
apply (frule l-dispose1-nonempty-above-singleton)
unfolding sep0-def
apply simp
unfolding F1-inv-def
apply (elim conjE bezE)
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(- UN m -)
thm k-min-loc-munion-singleton[of [d1 + s1 → the (f1 (d1 + s1))] UN 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-asn)
apply smt
by (metis l-dagger-not-empty map-upd-nonempty)

lemma \( z\text{-}\text{F1-inv-dispose1-finite} \):
\[
\text{finite } (\text{dom } ((\text{dom } \text{dispose1-below } f1 \text{ d1}) \cup \text{dom } (\text{dispose1-above } f1 \text{ d1 } s1)) - f1 \cup m \\
\text{[min-loc } (\text{dispose1-ext } f1 \text{ d1 } s1) \mapsto \text{HEAP1.sum-size } (\text{dispose1-ext } f1 \text{ d1 } s1)]
\]

find-theorems simp:finite(dom(-\cupm-))
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 \( z\text{-}\text{F1-inv-dispose1-nat1-map} \):
\[
\text{nat1-map } ((\text{dom } \text{dispose1-below } f1 \text{ d1}) \cup \text{dom } (\text{dispose1-above } f1 \text{ d1 } s1)) - f1 \cup m \\
\text{[min-loc } (\text{dispose1-ext } f1 \text{ d1 } s1) \mapsto \text{HEAP1.sum-size } (\text{dispose1-ext } f1 \text{ d1 } s1)]
\]

find-theorems simp:-\cupm-
find-theorems nat1-map [-\mapsto-]
apply (rule l-nat1-map-munion)
apply (metis F1-inv-def k-sep-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 \( z\text{-}\text{F1-inv-dispose1-sep} \):
\[
\text{sep } ((\text{dom } \text{dispose1-below } f1 \text{ d1}) \cup \text{dom } (\text{dispose1-above } f1 \text{ d1 } s1)) - f1 \cup m \\
\text{[min-loc } (\text{dispose1-ext } f1 \text{ d1 } s1) \mapsto \text{HEAP1.sum-size } (\text{dispose1-ext } f1 \text{ d1 } s1)]
\]

find-theorems sep (-\cupm-)
apply (rule l-sep-munion)
apply (metis l-dispose1-munion-disjoint)
apply (metis 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] F1-inv-finite)
apply (frule-tac [1-2] F1-inv-nat1-map)
apply (frule-tac [1-2] F1-inv-disjoint)
apply (frule-tac [1-2] F1-inv-sep)
by (simp-all add: l-sep0-dispose1-abovebelow-ext l-sep0-dispose1-ext-abovebelow)
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

lemma z-F1-inv-dispose1-Disjoint:
Disjoint
((dom (dispose1-below f1 d1) ∪ dom (dispose1-above f1 d1 s1)) ∪ m
  ⦵ f1 ⦵ m
  [min-loc (dispose1-ext f1 d1 s1) ↦ HEAP1.sum-size (dispose1-ext f1 d1 s1)])
find-theorems simp:Disjoint(- ∪ -)

apply (rule l-disjoint-singleton-upd)
apply (metis F1-inv-def l1-input-notempty-def l1-invariant-def l1-nat1-sum-size-dispose1-ext)
apply (metis F1-inv-def l1-dispose1-precondition-def l1-input-notempty-def l1-invariant-def l-disjoint-dispose1-ext)
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)) -\ a 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
unfolding 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]
E.9. PROOF OF SOME PROPERTIES OF INTEREST

unfolding $F_1$-inv-defs by auto

lemma $\neg F_1$-inv $[0 \mapsto 4, 4 \mapsto 11]$
unfolding $F_1$-inv-defs by auto

E.9.2 Operations properties

E.9.2.1 NEW 1 shrinks the memory

context level1-new
begin
$f_1' \subseteq_m f_1$ 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-asmp-simp) add: l-map-dom-ar-neq)
apply (insert l1-invariant-def)
apply (insert l1-input-notempty-def)
unfolding $F_1$-inv-def
apply (elim conjE)
apply (subgoal-tac dom $\{r\} - \{f_1\} \cap \text{dom } [r + s_1 \mapsto \text{the}(f_1 r) - s_1] = \{\}$)
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 $F_1$-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 $\{r\} - \{f_1\} \cap \text{dom } [r + s_1 \mapsto \text{the}(f_1 r) - s_1] = \{\}$)
— G1.2: new1_gr locs-subset (assuming munion WD)
apply (simp add: l-munion-dom l-munion-apply)
apply (rule 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
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

— G1.2.1: 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)
def er 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)
unfoldFing 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 ↦ (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)
E.9. PROOF OF SOME PROPERTIES OF INTEREST

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}.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 assumptions, 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-locs-singleton-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 - - U 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 (- U m)
find-theorems nat1-map [- -]
find-theorems nat1-map (- -)
find-theorems nat1-map (- -)
thm l-locs-munion-iff[of {r}.r U 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
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

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 (frule f-F1-inv-mapupd-keep-sep[of f1 r s1])

find-theorems simp:(- = False)
apply (insert k-new1-nat1-map-dom-ar[of r])
apply (frule 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
tatomize-not
l-locs-dom-ar-iff
l-locs-singleton-iff)

find-theorems ( - - - ) ∪ -
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})
- n ∉
(({{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 ⇒}
E.9. PROOF OF SOME PROPERTIES OF INTEREST

the \( (f r) - n \) \( l \) + \( n \)
apply (simp add: l-munion-apply l-munion-dom f-in-dom-ar-apply-subsume)
apply (cases \( l=m+n \))
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 \( r+n \mapsto (f r) - n \) , 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 \} -\alpha f) r = empty)
  thm l-sum-size-munion-singleton[simplified] f-d1-not-dispose-above[of n \{ r \} -\alpha f, simplified]
  apply (subl l-sum-size-munion-singleton)
  apply (metis k-finite-dispose-abovebelow-munion nat1-def)
  apply (smt Un-commute disjoint-iff-not-equal 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)
— case1.1.1 [3]: new1_eq; below=above=empty
  apply (cases dispose1-above (((\{ r \} -\alpha f) r n = empty)
  thm l-sum-size-munion-singleton[simplified] f-d1-not-dispose-above[of n \{ r \} -\alpha f, simplified]
  apply (metis k-finite-dispose-abovebelow-munion nat1-def)
  apply (smt Un-commute disjoint-iff-not-equal 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)
— case1.1.2 [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-abovebelow-munion nat1-def)
— case1.2 [5]: new1_eq; not below = empty
— case1.2.1 [6]: above=empty; not below = empty
  apply (cases dispose1-above (((\{ r \} -\alpha f) r n = empty)
  apply (simp add: l-dispose1-nonempty-above-singleton l-sum-size-singleton)
  apply (metis (full-types) k-empty-dispose-abovebelow-munion nat1-def)
apply simp
apply (erule bexE)
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

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
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)
E.9. PROOF OF SOME PROPERTIES OF INTEREST

unfolding dispose\text{-ext-def}

— case 2.1 [9]: new\textsubscript{1}\_gr; below=empty

apply (cases dispose\textsubscript{1}-below (\{r\} -\textcircled{a} f \cup m [r + n \mapsto \text{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\} -\textcircled{a} f, simplified]

apply (frule f-d1-not-dispose-above[of n r \{r\} -\textcircled{a} f \cup m [r + n \mapsto \text{the} (f r) - n]), simplified)

apply (subst l-munion-dom-ar-singleton)

apply (metis k-finite-dispose-above)

apply (simp add: l-munion-empty-lhs l-munion-empty-rhs)

— case 2.1.1 [10]: new\textsubscript{1}\_gr; below=above=empty

apply (simp-all add: l-munion-empty-lhs l-dom-ar-none l-sum-size-singleton)

thm l-munion-subsume l-dispose\textsubscript{1}-sep0-below-empty-iff[of - r n]

pr

apply (simp add: l-dispose\textsubscript{1}-sep0-above-empty-iff)

unfolding sep\textsubscript{0}-def

apply (simp)

apply (erule notE)

thm l-munion-dom[of \{r\} -\textcircled{a} f [r + n \mapsto \text{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 nat\_def)

apply (metis l-disjoint-mapupd-keep-sep l-dom-ar-notin-dom-or nat\_def)

— case 2.1.2 [11]: new\textsubscript{1}\_gr; below=empty; not above = empty

apply (simp add: l-dispose\textsubscript{1}-nonempty-above-singleton)

apply (simp add: l-sum-size-singleton)

thm l-munion\_apply[of \{r\} -\textcircled{a} f [r + n \mapsto \text{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-dispose\textsubscript{1}-sep0-above-empty-iff l-dispose\textsubscript{1}-sep0-below-empty-iff[of - r n])

unfolding sep\textsubscript{0}-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)

— case 2.2 [12]: new\textsubscript{1}\_gr; not below =empty

— case 2.2.1 [13]: new\textsubscript{1}\_gr; above=empty; not below = empty

apply (cases dispose\textsubscript{1}-above (\{r\} -\textcircled{a} f \cup m [r + n \mapsto \text{the} (f r) - n]) r n = empty)

apply (simp add: l-dispose\textsubscript{1}-sep0-above-empty-iff)

apply (unfold sep\textsubscript{0}-def)
apply (simp add: l-munion-dom)

— case 2.2.2 [14]: newl_gr; not above = empty; not below = empty
apply (simp add: l-dispose1-sep0-empty iff [of \(\{r\} -\# f \cup m \quad [r + n \mapsto \text{the} (f r) - n]\) r n])
apply (unfold sep0-def)
apply simp
apply (erule bexE)
find-theorems
thm k-min-loc-munion-singleton
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 (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:
\(0 < n \implies r \in dom f \implies \text{nat1-map} f \implies \text{Disjoint} f \implies r + n \in dom (\{r\} -\# f \cup m \quad [r + n \mapsto \text{the} (f r) - n]) \implies r + n \notin dom (\{r\} -\# f) \implies f = \{r + n\} -\# (\{r\} -\# f \cup m \quad [r + n \mapsto \text{the} (f r) - n]) \cup m \cup m \quad [r \mapsto \text{the} (f r)]\)
by (metis l-munion-dom-ar-singleton-subsume l-munion-subsume)

lemma l-new1-dispose-1-identity-case-11-original:
\(0 < n \implies r \in dom f \implies \text{nat1-map} f \implies \text{Disjoint} f \implies r + n \in dom (\{r\} -\# f \cup m \quad [r + n \mapsto \text{the} (f r) - n]) \implies r + n \notin dom (\{r\} -\# f) \implies f = \{r + n\} -\# (\{r\} -\# f \cup m \quad [r + n \mapsto \text{the} (f r) - n]) \cup m \cup m \quad [r \mapsto \text{the} (f r)]\)
find-theorems simp: -\# -\cup m -
thm l-munion-dom-ar-assoc[of \(\{r + n\} (\{r\} -\# f \cup m \quad [r + n \mapsto \text{the} (f r) - n]) \cup m \cup m \quad [r \mapsto \text{the} (f r)], simplified]
l-munion-subsume l-munion-assoc
k-new1-gr-dom-ar-dagger-aux[of - r n]
E.9. PROOF OF SOME PROPERTIES OF INTEREST

apply (subst l-munion-dom-ar-assoc[of \{r+n\} \{r\} \(-\emptyset\ f \cup m [r + n \mapsto \text{the } (f r) - n]\) [r \mapsto \text{the } (f r)]],simplified)
apply simp
apply (simp add: l-munion-dom l-dom-dom-ar)
thm l-munion-assoc[of \{r\} \(-\emptyset\ f \cup m [r + n \mapsto \text{the } (f r) - n]\) [r \mapsto \text{the } (f r)]]
apply (subst l-munion-assoc[of \{r\} \(-\emptyset\ f \cup m [r + n \mapsto \text{the } (f r) - n]\) [r \mapsto \text{the } (f r)]],simp-all)
thm l-munion-commute[of \{r\} \(-\emptyset\ f \cup m [r + n \mapsto \text{the } (f r) - n]\) [r \mapsto \text{the } (f r)]],simp
apply (subst l-munion-commute[of \{r\} \(-\emptyset\ f \cup m [r + n \mapsto \text{the } (f r) - n]\) [r \mapsto \text{the } (f r)]],symmetric)
apply (simp add: l-munion-dom-ar-assoc[of \{r+n\} f [r + n \mapsto \text{the } (f r) - n],simplified,symmetric]

find-theorems simp:dom(- -\emptyset -)
apply (simp add: l-dom-dom-ar)

thm b-dagger-munion[of \{r\} \(-\emptyset\ f \cup m [r + n \mapsto \text{the } (f r) - n]\),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:
\( r + n \notin \text{dom } \{\{r\} \{-\emptyset\ f \cup m \} \mapsto \text{the } (f r) - n\} \Rightarrow \)
\( r + n \notin \text{dom } \{\{r\} \{-\emptyset\ f \cup m \} \mapsto \text{the } (f r) - n\} \Rightarrow \)
\( l \in \text{dom } \{\{r\} \{-\emptyset\ f \cup m \} \mapsto \text{the } (f r) - n\} \Rightarrow \)
\( l + \text{the } \{\{\{r\} \{-\emptyset\ f \cup m \} \mapsto \text{the } (f r) - n\} \} l = r \Rightarrow \)
\( f = \{l\} \{-\emptyset\ \{\{r\} \{-\emptyset\ f \cup m \} \mapsto \text{the } (f r) - n\} \cup m \} l \mapsto \text{the } \{\{\{r\} \{-\emptyset\ f \cup m \} \mapsto \text{the } (f r) - n\} \cup m \} l \mapsto \text{the } (f r) - n\} l + n \)

by (simp add: l-munion-dom)

find-theorems locs -

thm l-locs-dom-ar-iff

\text{HEAP1SanityProofs.level1-new.new1-postcondition-state-locs-subset-algebraic}
l-disjoint-dispose1-ext

\text{HEAP1Proofs.level1-dispose.z-F1-inv-dispose1-Disjoint}

thm l-disjoint-singleton-upd

\text{HEAP1Proofs.level1-dispose.z-F1-inv-dispose1-Disjoint}

find-theorems locs - = locs -
APPENDIX E. HEAP LEMMAS AND PROOFS (LEO)

end
Appendix F

Heap lemmas and proofs (Iain)

theory HEAP0ProofsI JW
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 ∃· f′ r′. (is-block r′ s0 f0 ∧ f′ = f0 - locs-of r′ s0) ∧ 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: f0new = (f0 ∪ 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 ∪ 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 ∃· f′ f′ = f0 ∪ locs-of d0 s0 ∧ F0-inv f′ by blast
    qed
qed

end
theory HEAP1LemmasIJW
imports HEAP1
begin

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

lemma invF1-Disjoint-weaken: F1-inv f ⇒ Disjoint f
  unfolding F1-inv-def by simp

lemma invF1-nat1-map-weaken: F1-inv f ⇒ nat1-map f
  unfolding F1-inv-def by simp

lemma invF1-finite-weaken: F1-inv f ⇒ finite (dom f)
  unfolding F1-inv-def by simp

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 invVDMF1[intro!]: sep f ⇒ Disjoint f ⇒ VDM-F1-inv f
  unfolding VDM-F1-inv-def by simp

lemma ballUnE[elim!]: ∀· x ∈ f ∪ g. P x ⇒ (∀· x ∈ f. P x ⇒ ∀· x ∈ g. P x ⇒ R) ⇒ R
  by auto

lemma ballUnI[intro!]: ∀· x ∈ f. P x ⇒ ∀· x ∈ g. P x ⇒ ∀· x ∈ f ∪ g. P x
  by auto

lemma setminus-trans: X - insert x F = (X - F) - {x}
  by (metis Diff-insert)

lemma UN-minus: ∀· x ∈ X-{y}. P x ∩ P y = {x} ⇒ (∪· x ∈ X-{y}. P x) = (∪· x ∈ X. P x) - P y
  by blast

lemma UN-minus-gen:
  ∀· x ∈ X. ∀· y ∈ Y. P x ∩ P y = {x} ⇒ (∪· x ∈ X-Y. P x) = (∪· x ∈ X. P x) - (∪· y ∈ Y. P y)
  by blast

lemma union-comp: {x ∈ A ∪ B. P x} = {x ∈ A. P x} ∪ {x ∈ 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)
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

lemma $k$-in-locs-iff: $\text{nat1-map } f \implies (x \in \text{locs } f) = (\exists \cdot y \in \text{dom } f . x \in \text{locs-of } y \text{ (the}(f \ y)))$
unfolding locs-def
by (metis (mono-tags) UN-iff)

lemma $l$-locs-of-within-locs:
$\text{nat1-map } f \implies x \in \text{dom } f \implies \text{locs-of } x \text{ (the}(f\ x)) \subseteq \text{locs } f$
by (metis $k$-in-locs-iff subsetI)

lemma $b$-locs-of-as-set-interval:
$\text{nat1 } n \implies \text{locs-of } l\ n = \{l..l+n\}$
unfolding locs-of-def
by (metis Collect-conj-eq atLeastLessThan-def atLeast-def lessThan-def)

lemma locs-of-subset:
$\text{nat1 } m - s \implies \text{locs-of } l\ m - (m - s) \subseteq \text{locs-of } l\ m$
apply (subt $b$-locs-of-as-set-interval, simp)
apply (subt $b$-locs-of-as-set-interval, simp)
by simp

lemma domf-in-locs:
$\text{nat1-map } f \implies \text{dom } f \subseteq \text{locs } f$
unfolding locs-def
apply simp
by (metis locs-def not-dom-not-locs-weaken subsetI)

lemma locs-of-finite:
$\text{nat1 } s \implies \text{finite } \text{locs-of } l\ s$
unfolding locs-of-def
by auto

lemma l-dom-in-locs-of:
$\text{nat1-map } f \implies x \in \text{dom } f \implies x \in \text{locs-of } x \text{ (the}(f\ x))$
apply (subt $b$-locs-of-as-set-interval)
apply (simp add: nat1-map-def)
apply (simp add: nat1-map-def)
done

lemma locs-of-unique:
$\text{nat1 } y \implies \text{nat1 } y' \implies \text{locs-of } x\ y = \text{locs-of } x\ y' \implies x = x' \land y = y'$
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:
$\text{nat1 } y \implies \text{nat1 } y' \implies \text{locs-of } x\ y = \text{locs-of } x\ y' = (y = y')$
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:
$\text{nat1-map } f \implies \text{nat1-map } g \implies x \in \text{dom } f \implies x' \in \text{dom } g \implies \text{locs-of } x \text{ (the}(f\ x)) = \text{locs-of } x' \text{ (the}(g\ x')) \implies x = x'$
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: \( x - 1 \in \text{locs-of} \ a \ b \implies x \notin \text{locs-of} \ a \ b \implies \text{nat1} \ b \implies x = a + b \)
  by (auto simp add: b-locs-of-as-set-interval)

lemma locs-empty: \( \text{locs empty} = \{\} \) unfolding locs-def
by (metis SUP-empty dom-empty empty-iff nat1-map-def)

lemma empty-locs-empty-map: \( \text{nat1-map} \ f \implies \text{locs} \ f = \{\} \implies f = \text{empty} \)
  unfolding locs-def apply simp
by (metis domIff empty-iff l-dom-in-locs-of)

lemma locs-of-pred: \( x \neq a \implies \text{nat1} \ b \implies x \in \text{locs-of} \ a \ b \implies x - 1 \in \text{locs-of} \ a \ b \)
  apply (simp add: b-locs-of-as-set-interval) by auto

lemma locs-of-pred2: assumes xgr0: \( x > 0 \) and nat1f: \( \text{nat1-map} \ f \)
  and minusone: \( x - 1 \in \text{locs-of} \ a \ (\text{the} \ (f \ a)) \)
  and xindom: \( x \in \text{dom} \ f \) and aindom: \( a \in \text{dom} \ f \)
  and Disj: \( \text{Disjoint} \ f \)
  shows \( x \notin \text{locs-of} \ a \ (\text{the} \ (f \ a)) \)
proof -
  have \( x \in \text{locs-of} \ x \ (\text{the} \ (f \ x)) \) by (metis \( l\)-dom-in-locs-of nat1f xindom)
  from Disj have \( \text{locs-of} \ x \ (\text{the} \ (f \ x)) \cap \text{locs-of} \ a \ (\text{the} \ (f \ a)) = \{\} \)
  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 simpE)
  apply (rule notI)
  proof -
    assume \( x = a \)
    then have \( *: x - 1 \in \text{locs-of} \ x \ (\text{the} \ (f \ x)) \) by (metis minusone)
    have \( **: x - 1 \notin \text{locs-of} \ x \ (\text{the} \ (f \ x)) \)
      apply (subst b-locs-of-as-set-interval)
      apply (metis nat1-map-def nat1f xindom)
    using xgr0 by auto
    from \( * \ \star \star \) show False by auto
  next
    assume \( \text{locs-of} \ x \ (\text{the} \ (f \ x)) \cap \text{locs-of} \ a \ (\text{the} \ (f \ a)) = \{\} \)
    then show \( \text{locs-of} \ x \ (\text{the} \ (f \ x)) \cap \text{locs-of} \ a \ (\text{the} \ (f \ a)) = \{\} \)
      by simp
  next
    assume \( a \notin \text{dom} \ f \)
    then show \( \text{locs-of} \ x \ (\text{the} \ (f \ x)) \cap \text{locs-of} \ a \ (\text{the} \ (f \ a)) = \{\} \) using aindom by auto
  next
    assume \( x \notin \text{dom} \ f \) then show \( \text{locs-of} \ x \ (\text{the} \ (f \ x)) \cap \text{locs-of} \ a \ (\text{the} \ (f \ a)) = \{\} \)
      using xindom by simp
  qed
  then show \( x \notin \text{locs-of} \ a \ (\text{the} \ (f \ a)) \)
    by (metis \( x \in \text{locs-of} \ x \ (\text{the} \ (f \ x)) \); disjoint-iff-not-equal)
  qed

lemma locs-of-extended: \( \exists \ y \in \text{locs-of} \ x \ a \ y \notin \text{locs-of} \ x \ b \implies \text{nat1} \ a \implies \text{nat1} \ b \implies a > b \)
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 lidom: l ∈ dom f
and flbigger: 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 flbigger 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 lidom nat1-map-def)
then show thesis unfolding locs-of-def
by (simp add: flbigger)
qed
have notl: l+s ≠ l using nat1s by auto
havenotinlocs: 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 lidom lsindom natl)
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 ≠ empty and notempg: g ≠ empty
shows f = g
proof -
  have dom-eq: dom f = dom g
  proof (rule ccontr)
    assume doms-not-equal: dom f ≠ dom g
    have elem-in-fnotg-or-gnotf: (∃· x ∈ dom f. x /∈ dom g) ∨ (∃· x ∈ dom g. x /∈ dom f)
      by (metis (full-types) doms-not-equal subsetI subset-antisym)
    then show False
      proof (elim bexE disjE)
        fix x
        assume xinf: x ∈ dom f and xnoting: x /∈ dom g
        show False
          proof (cases x > 0)
            assume xgr0: x > 0
            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 locs-eq)
            then obtain y where ying: y ∈ dom g and xlocsofy
              x ∈ locs-of y (the (g y))
              by auto
            from ying xlocsofy have x ≠ y by (metis xnoting)
            then have x - 1 ∈ locs-of y (the (g y))
              by (metis invF1-nat1-map-weaken invg k-in-locs-iff)
            then have x - 1 ∈ locs g by (metis invF1-nat1-map-weaken invg k-in-locs-iff)
            then have xminus1-in-locsf: x - 1 ∈ 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 /∈ locs f
              proof
                let ?x’ = x - 1
                assume xminusinlocs: ?x’ ∈ locs f
                then have ∃· below ∈ dom f. ?x’ ∈ locs-of below (the (f below))
                  by (metis invF1-nat1-map-weaken invf k-in-locs-iff)
                then obtain below where belowsf: below ∈ dom f
                  and locsofbelow: ?x’ ∈ locs-of below (the (f below))
                    by auto
                have x ∈ dom f by (metis xinf)
                have notlocsofx: x /∈ locs-of below (the (f below))
                  by (metis invF1E belowsf invf locs-of-pred2 locsofbelow xgr0 xinf)
                from locsofbelow notlocsofx have x = below + the (f below)
                  by (metis belowsf 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-locs f)
next
assume ¬x >0
then have xzero: x = 0 by (metis neq0-conv)
have x ∈ locs f (the (f x))
  by (metis invF1-nat1-map-weaken invf l-dom-in-locs-of xinf)
then have x ∈ locs g by (metis locs-eq)
then have x ∈ locs f by (metis invF1-nat1-map-weaken invg 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 zlocsofy: 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 k-locs-of-as-set-interval invF1-nat1-map-weaken invf nat1-map-def ying)
then have x ∈ {y..<y + (the (g y))} by (metis zlocsofy)
then have zeqy: x = y using xzero by auto
from ynoteqx and zeqy 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 f (the (f x))
    by (metis invF1-nat1-map-weaken invf not-dom-not-locs-weaken xinf)
then have x ∈ locs g by (metis locs-eq)
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 zlocsofy: x ∈ locs-of y (the (f y))
  by auto
from ying zlocsofy have x ≠ y by (metis xnotinf)
then have x - 1 ∈ locs f (the (f y))
  by (metis invF1-nat1-map-weaken invf locs-of-pred nat1-map-def zlocsofy ying)
then have x - 1 ∈ locs f by (metis invF1-nat1-map-weaken invf k-in-locs-iff ying)
then have xminus1: x - 1 ∈ locs g by (metis locs-eq)
from invg have sepq: sep g by (rule invF1-sep-weaken)
from invg have Disjg: Disjoint g by (rule invF1-Disjoint-weaken)
then have x - 1 /∈ locs g
proof
  let ?x' = x - 1
  assume xminusinloc: ?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 belowofbelow: ?x' ∈ locs-of below (the (g below))
    by auto
have \( x \in \text{dom } g \) by (metis xing)
have notlocsofx: \( x \notin \text{locs-of } \text{below} \) \((g \text{ below})\)
by (metis Disjg HEAP1 LemmasI JW.invF1-nat1-map-weaken belowing invg locs-of-pred2 locsofbelow xgr0 xing)
from locsofbelow notlocsofz have \( x = \text{below } + \) \((g \text{ below})\)
thus False by (metis belowing sep-def sepg xing)

qed

thus False by (metis xminus1ing)

next

assume \( \neg x > 0 \)
then have xzero: \( x = 0 \) by (metis neq0-conv)
have \( x \in \text{locs-of } \text{the} \) \((g \text{ x})\)
by (metis invF1-nat1-map-weaken invg l-dom-in-locs-of xing)
then have \( x \in \text{locs } f \) by (metis locs-eq)
then have \( x \in \text{locs } f \) by (metis locs-eq)

then obtain \( y \) where \( y \neq x \) by (metis xnotinf yinf)

have \( \text{locs-of } \text{y } \text{the} \) \((f y)\) by (metis invF1-nat1-map-weaken infv k-in-locs-iff)
then obtain \( y \) where \( y \in \text{dom } f \) \text{and} \( x \in \text{locs-of } \text{y } \text{the} \) \((f y)\)
by auto
have ynotx: \( y \neq x \) by (metis xnotinf yinf)

then \( \text{ylocsofy} \)
then have \( x \in \text{locs } f \) by (metis locs-eq)

then \( \text{yeqy} \)
then have xeqy: \( x = y \) using xzero by auto
from ynotx and xeqy show False by simp

dq
dq
dq

dq

show ?thesis

proof
fix \( x \)
show \( f x = g x \)
proof (cases \( x \in \text{dom } f \))
assume xinf: \( x \in \text{dom } f \)
show ?thesis
proof (cases \( x \in \text{dom } g \))
assume xing: \( x \in \text{dom } g \)
have (the \((f x)\)) = (the \((g x)\))
proof -

have natinfx: \( \text{nat1 } \) \((f x)\) by (metis invF1-nat1-map-weaken dom-eq invf nat1-map-def xing)
have nat1gx: \( \text{nat1 } \) \((g x)\) by (metis invF1-nat1-map-weaken dom-eq invg nat1-map-def xing)

have \( \text{locs-of } \text{the} \) \((f x)\) = \( \text{locs-of } \text{the} \) \((g x)\)
proof (rule ccontr)
assume \( \text{locs-of-f-not-g } \) \((f x)\) \( \neq \) \( \text{locs-of } \) \((g x)\)

then have \( \exists \cdot \text{x } \text{y } \text{the} \) \((f x)\) \( \neq \) \((g x)\)

by auto
from this show False

proof
assume \( \exists \cdot \text{y } \text{locsof} \) \((f x)\) \( \neq \) \( \text{locs-of } \) \((g x)\)
then have fgrg: \( \text{the} \) \((f x)\) > \( \text{the} \) \((g x)\)
by (metis locs-of-extended nat1fx nat1gx)

have firstpartcontr: \( x + \{ (g x) \} \notin \text{dom } g \)

by (metis invF1-sep-weaken invg sep-def xing)

then have \( x + \{ (g x) \} \in \text{locs-of } x \text{ (the } (f x) \) \)

by (metis b-locs-of-as-set-interval nat1fx nat1gx locs-of-f-not-g

fng locs-of-edge top-locs-of2)

then have \( x + \{ (g x) \} \in \text{locs } f \)

by (metis invF1-nat1-map-weaken invf k-in-locs-iff xing)

then have \( \exists \cdot x + \{ (g x) \} \in \text{locs } \text{of } \text{loc } \text{ (the } (g \text{ loc}) \) \)

by (metis invF1-nat1-map-weaken invg k-in-locs-iff)

then obtain loc where locing: \( \text{loc } \in \text{dom } g \) and \( x + \{ (g x) \} \in \text{locs-of } \text{loc } \text{ (the } (g \text{ loc}) \)

by auto

have \( x + \{ (g x) \} - 1 \in \text{locs-of } x \text{ (the } (g x) \) \)

by (metis nat1gx top-locs-of)

have locnotg: \( \text{loc } \neq x \)

proof

assume locexq: \( \text{loc } = x \)

then have \( \text{loc } + \{ (g \text{ loc}) \} \in \text{locs-of } \text{loc } \text{ (the } (g \text{ loc}) \) \)

by (metis \( \{ x \} + \{ (g x) \} \in \text{locs-of } \text{loc } \text{ (the } (g \text{ loc}) \) \)

moreover have \( \text{loc } + \{ (g \text{ loc}) \} \notin \text{locs-of } \text{loc } \text{ (the } (g \text{ loc}) \) \)

using b-locs-of-as-set-interval by (simp del: nat1-def add: nat1-map-def nat1gx locexq)

ultimately show False by simp

qed

from invg have Disjoint g by (rule invF1-Disjoint-weaken)

then have \( \text{locs-of } \text{loc } \text{ (the } (g \text{ loc}) \) \cap \text{locs-of } x \text{ (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 \( \text{loc } = x + \{ (g x) \) \)

by (metis (hide-lams, full-types) F1-inv-def \( \{ x + \{ (g x) \} - 1 \in \text{locs-of } x \text{ (the } (g x) \) \)

\( \{ x + \{ (g x) \} \in \text{locs-of } \text{loc } \text{ (the } (g \text{ loc}) \) \cap \text{locs-of } x \text{ (the } (g x) \) \)

= \{ \})

comm-monoid-add-class.add.right-neutral disjoint-iff-not-equal dom-eq inf.commute

invg locing

\( \text{locs-of-pred } \text{nat1-def neq0-conv sep-def} \)

then have \( x + \{ (g x) \} \in \text{dom } g \) by (metis locing)

thus False using firstpartcontr by auto

next

assume \( \exists \cdot y \in \text{locs } x \text{ (the } (g x) \), \( y \notin \text{locs-of } x \text{ (the } (f x) \) \)

then have ggrf: \( (g x) > (f x) \) by (metis (full-types) locs-of-extended nat1fx nat1gx)

have firstpartcontr: \( x + \{ (f x) \} \notin \text{dom } f \)

by (metis invF1-sep-weaken dom-eq invf sep-def xing)

then have \( x + \{ (f x) \} \in \text{locs } f \text{ (the } (f x) \) \)

by (metis ggrf b-locs-of-as-set-interval locs-of-edge locs-of-f-not-g nat1fx nat1gx top-locs-of2)

then have \( x + \{ (f x) \} \in \text{locs } g \)

by (metis invF1-nat1-map-weaken invg k-in-locs-iff xing)

then have \( x + \{ (f x) \} \in \text{locs } f \) by (metis locs-eq)
then have \( \exists \cdot \, \text{loc} \in \text{dom} \, f \cdot x + \text{the} \, (f \, x) \in \text{locs-of loc} \, (\text{the} \, (f \, \text{loc})) \)
by (metis invF1-nat1-map-weaken inf-k-m-locs-iff)
then obtain \text{loc} \ where \ locinf: \text{loc} \in \text{dom} \, f \ \text{and} \ x + \text{the} \, (f \, x) \in \text{locs-of loc} \, (\text{the} \, (f \, \text{loc}))
by auto
have \( x + \text{the} \, (f \, x) - 1 \in \text{locs-of x} \, (\text{the} \, (f \, x)) \)
by (metis nat1fx top-locs-of)
have \( \text{locnotg} : \text{loc} \not= x \)
proof
\begin{proof}
assume \( \text{loceqx} : \text{loc} = x \)
then have \( \text{loc} + \text{the} \, (f \, \text{loc}) \in \text{locs-of loc} \, (\text{the} \, (f \, \text{loc})) \)
by (metis \( x + \text{the} \, (f \, x) \in \text{locs-of loc} \, (\text{the} \, (f \, \text{loc})))\)
moreover have \( \text{loc} + \text{the} \, (f \, \text{loc}) \not\in \text{locs-of loc} \, (\text{the} \, (f \, \text{loc})) \)
by (simp del: nat1-def add: b-locs-of-as-set-interval nat1fx \( \text{loceqx} \))
ultimately show \( \text{False} \)
by simp
qed
\end{proof}
then have \( \text{Disjoint} \, f \)
by (metis invF1-Disjoint-weaken)
then have \( \text{locs-of loc} \, (\text{the} \, (f \, \text{loc})) \cap \text{locs-of x} \, (\text{the} \, (f \, x)) = \{\} \)
unfolding Disjoint-def Locs-of-def
apply (simp add: locinf)
apply (erule-tac \( x \in \text{ballE} \))
apply (erule-tac \( x = x \in \text{ballE} \))
apply (erule impE)
apply (rule locnotg)
apply (metis disjoint-def)
apply (simp add: xinf)
by (simp add: locinf)
have \( \text{loc} = x + \text{the} \, (f \, x) \)
by (metis (hide-lams, full-types) F1-inv-def \( x + \text{the} \, (f \, x) - 1 \in \text{locs-of x} \, (\text{the} \, (f \, x)) \))
\( x + \text{the} \, (f \, x) \in \text{locs-of loc} \, (\text{the} \, (f \, \text{loc})) \cap \text{locs-of x} \, (\text{the} \, (f \, x)) \)
= \{\}
comm-monoid-add-class.add.right-neutral disjoint-iff-not-equal dom-eq inf.commute inf
locinf
locs-of-pred nat1-def neq0-conv sep-def)
then have \( x + \text{the} \, (f \, x) \in \text{dom} \, f \)
by (metis locinf)
thus \( \text{False} \)
using firstpartcontr by auto
qed
next
assume \( \text{notg} : x \not\in \text{dom} \, g \)
then have \( \text{notf} : x \not\in \text{dom} \, f \)
using dom-eq by simp
from \( \text{notg} \, \text{notf} \)
show \( \text{?thesis} \)
by auto
qed
next
assume \( \text{xnotf} : x \not\in \text{dom} \, f \)
then have \( x \not\in \text{dom} \, g \)
using dom-eq by simp
thus \( \text{?thesis} \)
using \( \text{xnotf} \) by auto
qed
qed
next
assume \( \text{xnotg} : x \not\in \text{dom} \, g \)
then have \( x \not\in \text{dom} \, f \)
thus \( \text{?thesis} \)
using \( \text{xnotg} \) by auto
qed
qed

lemma locs-singleton:
assumes \( * : \text{nat1} \, y \)
shows \( \text{locs }[x \mapsto y] = \text{locsof } x y \)

proof -
from * have nat1-map \([x \mapsto y]\]
by (metis dom-empty empty-iff fun-upd-same l-inmapupd-dom-iff nat1-map-def the.simps)
then show \( ?\text{thesis unfolding locs-def by simp }\)
qed

lemma locs-of-subset-range: \( x > 0 \implies y > 0 \implies \text{locsof } l \ x \subseteq \text{locsof } l \ y \implies y \geq x \)
by (simp add: b-locs-of-as-set-interval)

lemma locs-of-subset-range-gr: 
shows \( x > 0 \implies y > 0 \implies l > l' \implies \text{locsof } l \ x \subseteq \text{locsof } l' \ y \implies y \geq x \)
by auto

lemma less-a-not-in-locs-of: \( b > 0 \implies a > 0 \implies l \notin \text{locsof } a \ b \)
apply (subst b-locs-of-as-set-interval)
apply simp
by simp

lemma after-locs-of-not-in-locs: assumes invf: \( F1 \text{-inv } f1 \)
and mindom: \( m \in \text{dom } f1 \)
shows \( m + (\text{the } (f1 \ m)) \notin \text{locsof } f1 \)
proof
assume \( m + (\text{the } (f1 \ m)) \in \text{locsof } f1 \)
then have \( 3 \cdot n \in \text{dom } f1 \cdot m + (\text{the } (f1 \ m)) \in \text{locsof } n \ (\text{the } (f1 \ n)) \)
by (metis invF1-nat1-map-weaken invf k-in-locs-iff)
then obtain \( n \) where mindom: \( n \in \text{dom } f1 \) and
locsofn: \( m + (\text{the } (f1 \ m)) \in \text{locsof } n \ (\text{the } (f1 \ n)) \)
by auto
have \( m + (\text{the } (f1 \ m)) \notin \text{locsof } m \ (\text{the } (f1 \ m)) \)
apply (rule edge-not-in-locs-of) by (metis invF1-sep-weaken comm-monoid-add-class.add_right_neutral
injf mindom neq0-cone sep-def)
then have \( m \neq n \) by (metis locsofn)
have sep \( f1 \) by (metis invF1-sep-weaken invf)
then have \( m + (\text{the } (f1 \ m)) \notin \text{dom } f1 \) by (metis mindom sep-def)
moreover have \( m + (\text{the } (f1 \ m)) \in \text{dom } f1 \)
proof (rule ccontr)
assume \( m + (\text{the } (f1 \ m)) \notin \text{dom } f1 \)
have \( m + (\text{the } (f1 \ m)) \in \text{locsof } n \ (\text{the } (f1 \ n)) \) by (metis locsofn)
then have \( m + (\text{the } (f1 \ m)) - 1 \in \text{locsof } n \ (\text{the } (f1 \ n)) \)
by (metis invF1-nat1-map-weaken \( m + (\text{the } (f1 \ m)) \notin \text{dom } f1 \) invf locsof-pred nat1-map-def mindom)
moreover have \( m + (\text{the } (f1 \ m)) - 1 \in \text{locsof } m \ (\text{the } (f1 \ m)) \)

\[157\]
by (metis invF1-nat1-map-weaken infv mindom nat1-map-def top-locs-of)
moreover have Disjoint f1 by (metis invF1-Disjoint-weaken infv)
moreover have locs-of n (the (f1 n)) ∩ locs-of m (the (f1 m)) = {} by (metis Disjoint-def ⟨m ≠ n⟩)
ultimately show False by auto
qed
ultimately show False by auto
qed

lemma locs-of-boundaries: \( b > 0 \implies l \in \text{locs-of}\ a\ b \implies l \geq a \land l < a+b \)
by (simp add: b-locs-of-as-set-interval)

lemma locs-locs-of-subset:
assumes subset: \( \text{locs-of}\ l s1 \subseteq \text{locs}\ f1\)
and invf: \( F1\-inv f1\)
and nat1s1: \( \text{nat1}\ s1\)
shows \( \exists\ m \in \text{dom}\ f1. \text{locs-of}\ l s1 \subseteq \text{locs-of}\ m\ (\text{the}\ (f1\ m))\)
proof -

have \( l \in \text{locs-of}\ l s1\) using nat1s1
by (simp add: b-locs-of-as-set-interval)
then have \( l \in \text{locs}\ f1\) using subset by auto
then have \( l \in (\bigcup s \in \text{dom}\ f1. \text{locs-of}\ s\ (\text{the}\ (f1\ s)))\)
unfolding locs-def Locs-of-def
by (simp add: invf invF1-nat1-map-weaken)

have \( \exists\ m \in \text{dom}\ f1. l \in \text{locs-of}\ m\ (\text{the}\ (f1\ m))\)
by (metis invF1-nat1-map-weaken l-locs-of-Locs-of-iff mindom nindom)
then obtain \( m\) where mindom: \( m \in \text{dom}\ f1\) and

llocsof: \( l \in \text{locs-of}\ m\ (\text{the}\ (f1\ m))\)
by auto

have \( l+s1 - 1 \in \text{locs-of}\ l s1\)
by (simp nat1s1 top-locs-of)
then have \( l+s1 - 1 \in \text{locs}\ f1\)
by (metis set-map subset)
then have \( \exists\ n \in \text{dom}\ f1. l+s1 - 1 \in \text{locs-of}\ n\ (\text{the}\ (f1\ n))\)
by (metis invF1-nat1-map-weaken infv k-in-locs-iff)
then obtain \( n\) where nindom: \( n \in \text{dom}\ f1\) and

lplusinlocsof: \( l+s1 - 1 \in \text{locs-of}\ n\ (\text{the}\ (f1\ n))\)
by auto

have meqn: \( m = n\)
proof (rule ccontr)
assume noteq: \( m \neq n\)
then have \( m + (\text{the}\ (f1\ m)) \in \text{locs-of}\ l s1\)
proof -

have \( m \leq l\) by (metis invF1-sep-weaken comm-monoid-add-class.add.right-neutral infv linlocsof locs-of-boundaries mindom neq0-conv sep-def)
moreover have \( l < m + (\text{the}\ (f1\ m))\) by (metis invF1-nat1-map-weaken infv linlocsof locs-of-boundaries mindom nat1-def nat1-map-def)
moreover have \( n \leq l + s1 - 1\) by (metis invF1-sep-weaken comm-monoid-add-class.add.right-neutral infv locs-of-boundaries lplusinlocsof neq0-conv nindom sep-def)
moreover have \( l + s1 - 1 < n + (\text{the}\ (f1\ n))\) by (metis invF1-sep-weaken comm-monoid-add-class.add.right-neutral infv locs-of-boundaries lplusinlocsof neq0-conv nindom sep-def)
moreover have \( m + \text{the}(f1\ m) \leq n\)
proof (rule ccontr)
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

assume *: \( \neg m + \text{(f1} m) \leq n \)
then have **: \( n \leq m + \text{(f1} m) \) by (metis not-less)
moreover have \( n \geq m \)
by (smt \( l + s1 - 1 < n + \text{(f1} n) \); \( m \leq l \) invf mindom minor-sep-prop nat1-def nat1s1 nindom)

moreover have ***: \( n \in \text{locs-of} \text{(f1} m) \)
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 \( \text{locs-of} \text{(f1} n) \cap \text{locs-of} \text{(f1} m) = \) \( \emptyset \)
by (smt *** \( l < m + \text{(f1} m) \); \( m \leq l \) invf less-a-not-in-locs-of mindom minor-sep-prop nindom noteq)

moreover have \( n \in \text{locs-of} \text{(f1} n) \)
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 \( m + \text{(f1} m) \notin \text{locs} \text{f1} \)
by (metis after-locs-of-not-in-locs invf mindom)
ultimately show False by (metis in-mono subset)
qed

have \( \text{locs-of} \text{l} s1 \subseteq \text{locs-of} \text{(f1} m) \)
proof (rule locs-of-subset-top-bottom)
  show \( 0 < s1 \) by (metis nat1-def nat1s1)
next
  show \( 0 < \text{the} \text{(f1} m) \) by (metis invF1-sep-weaken invf mindom
  monoid-add-class.add.right-neutral neq0-conv sep-def)
next
  show \( l \in \text{locs-of} \text{(f1} m) \) by (rule linlocsof)
next
  show \( l + s1 - 1 \in \text{locs-of} \text{(f1} m) \) 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 \( (s \circ f) \)
unfolding nat1-map-def dom-antequiv-def
using * nat1-map-def by (simp add: domIff)

159
lemma dagger-nat1-map:
  nat1-map f \implies nat1-map g \implies nat1-map (f \dagger 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 \cap dom g = {} \implies nat1-map f \implies nat1-map g \implies nat1-map (f \cup m g)
unfolding union-def
  by (simp add: dagger-nat1-map)

lemma unionm-singleton-nat1-map:
  assumes disjdom: a \notin dom f
  and nat1f: nat1-map f
  and nat1b: nat1 b
  shows nat1-map (f \cup m [a \mapsto b])
proof (rule unionm-nat1-map)
  show nat1-map f by (rule nat1f)
next
  show nat1-map [a \mapsto b] using nat1b by (simp add: nat1-map-def)
  using disjdom by simp
qed

lemma locs-of-sum-range: nat1 y \implies nat1 z \implies locs-of x \(y+z\) = (locs-of x y) \cup (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 -\alpha f))
proof (rule finite-subset)
  show dom (s -\alpha f) \subseteq 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 \dagger g))
  by (simp add: l-dagger-dom *)

lemma dagger-singleton-finite:
  assumes *: finite (dom f)
shows finite \((\text{dom } (f \uparrow [a \mapsto b]))\)
by \((\text{simp add: l-dagger-dom } *)\)

\[\text{lemma \ unionm-finite:}\]
assumes \(\text{disjdom}: \text{dom } f \cap \text{dom } g = \{\}\) and \(*: \text{finite } (\text{dom } f) \text{ finite } (\text{dom } g)\)
shows \(\text{finite } (\text{dom } (f \cup m g))\)
by \((\text{metis } * \text{l-dagger-dom disjdom finite-UnI unionm-def})\)

\[\text{lemma \ unionm-singleton-finite:}\]
assumes \(\text{disjdom}: a \notin \text{dom } f\)
and \(*: \text{finite } (\text{dom } f)\)
shows \(\text{finite } (\text{dom } (f \cup m [a \mapsto b]))\)
by \((\text{simp add: unionm-finite } \text{disjdom})\)

\[\text{lemma \ dom-ar-sep:}\]
assumes \(*: \text{sep } f\)
shows \(\text{sep } (s -a f)\)
by \((\text{smt } * \text{f-in-dom-ar-subsume sep-def f-in-dom-ar-the-subsume})\)

\[\text{lemma \ singleton-sep: } \text{nat1 } b \implies \text{sep } [a \mapsto b]\]
unfolding \(\text{sep-def}\) by \(\text{simp}\)

\[\text{lemma \ dagger-singleton-sep:}\]
assumes \(*: \text{sep } f\)
and \(***: \forall \cdot l \in \text{dom } f. l + \text{the } ((f \uparrow [a \mapsto b]) l) \notin \text{dom } ([a \mapsto b])\)
and \(*****: a + b \notin \text{dom } f\)
and \(\text{anotinf}: a \notin \text{dom } f\)
and \(\text{nat1b: nat1 } b\)
shows \(\text{sep } (f \uparrow [a \mapsto b])\)
unfolding \(\text{sep-def}\)
proof \((\text{subst l-dagger-dom, rule ballUnI})\)
show \(\forall l \in \text{dom } f. l + \text{the } ((f \uparrow [a \mapsto b]) l) \notin \text{dom } (f \uparrow [a \mapsto b])\)
by \((\text{metis } *** \text{anotinf dagger-def domIff fun-upd-apply map-add-empty map-add-upd sep-def})\)
next
show \(\forall l \in \text{dom } [a \mapsto b]. l + \text{the } ((f \uparrow [a \mapsto b]) l) \notin \text{dom } (f \uparrow [a \mapsto b])\)
by \((\text{smt singleton-sep nat1b **** dagger-def domIff fun-upd-same l-inmapupd-dom-Iff map-add-None map-add-dom-app-simps(1) sep-def the.simps})\)
qed

\[\text{lemma \ unionm-singleton-sep:}\]
assumes \(\text{disjoint-dom}: a \notin \text{dom } f\)
and \(*: \text{sep } f\)
and \(***: \forall l \in \text{dom } f. l + \text{the } ((f \uparrow [a \mapsto b]) l) \notin \text{dom } ([a \mapsto b])\)
and \(*****: a + b \notin \text{dom } f\)
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 -o- 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 disj: 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 (rule-tac g=[a ↦→ b] in unionm-in-dom-left, rule xindom, simp add: disj anotinf)

moreover have (locs-of y (the ((f ∪ m [a ↦→ b]) y))) = (locs-of y (the (f y)))

by (rule-tac g=[a ↦→ b] in unionm-in-dom-left, simp 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)

from disj have disjoint (locs-of x (the (f x))) (locs-of a b)

proof (rule 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 (rule-tac g=[a ↦→ b] in unionm-in-dom-left, simp add: disj anotinf)

moreover have (locs-of y (the ((f ∪ m [a ↦→ b]) y))) = (locs-of a b)

by (rule-tac g=[a ↦→ b] in unionm-in-dom-left, simp add: yeqa anotinf)

ultimately show thesis by simp

next

assume ynoteqa: y ≠ a

then show thesis
proof

have xinf: \( x \in dom \; f \)
  \hspace{1em} by (rule-tac \( g=[a \mapsto b] \) in unionm-in-dom-left,
  \hspace{1em} rule xindom, simp add: disj anotinf, simp add: xnoteq)

have yinf: \( y \in dom \; f \)
  \hspace{1em} by (rule-tac \( g=[a \mapsto b] \) in unionm-in-dom-left,
  \hspace{1em} rule yindom, simp add: disj anotinf, simp add: ynoteq)

have disjoint (locs-of \( x \) (the (\( f \; x \)))) (locs-of \( y \) (the (\( f \; y \))))
  \hspace{1em} by (metis Disjoint-def xinf yinf disjf l-locs-of-Locs-of-iff xnoty)

moreover have (locs-of \( x \) (the ((\( f \cup m \; [a \mapsto b] \)) \( x \)))) = (locs-of \( x \) (the (\( f \; x \))))
  \hspace{1em} by (subst l-the-map-union-left, simp-all add: xnoteq xinf anotinf)

moreover have (locs-of \( y \) (the ((\( f \cup m \; [a \mapsto b] \)) \( y \)))) = (locs-of \( y \) (the (\( f \; y \))))
  \hspace{1em} by (subst l-the-map-union-left, simp-all add: ynoteq yinf anotinf)

ultimately show \( \exists \text{thesis by simp} \)

qed

qed

thus disjoint (Locs-of (\( f \cup m \; [a \mapsto b] \)) \( x \)) (Locs-of (\( f \cup m \; [a \mapsto b] \)) \( y \))

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: \( r \in dom \; f \)
  shows \( \text{locs}(\{r\} \mapsto f) = \text{locs} \; f \; - \; \text{locs-of} \; r \; (\text{the}\; (f \; r)) \)

proof

  have nat1-ar: nat1-map (\( \{r\} \mapsto f \)) using nat1f by (rule dom-ar-nat1-map)
  have \( (\bigcup s \in dom \; (\{r\} \mapsto f). \; \text{locs-of} \; s \; (\text{the}\; ((\{r\} \mapsto f) \; s))) = \)
    \( (\bigcup s \in dom \; ((\{r\} \mapsto f). \; \text{locs-of} \; s \; (\text{the}\; (f \; s)))) \)
    by (simp add: f-in-dom-ar-the-subsume)
  also have \( \ldots = (\bigcup s \in dom \; f - \{r\}. \; \text{locs-of} \; s \; (\text{the}\; (f \; s))) \)
    by (metis l-dom-dom-ar)
  also have \( \ldots = (\bigcup s \in dom \; f. \; \text{locs-of} \; s \; (\text{the}\; (f \; s))) - \text{locs-of} \; r \; (\text{the}\; (f \; r)) \)
    proof (rule UN-minus)
      show \( \forall \; x \in dom \; f - \{r\}. \; \text{locs-of} \; x \; (\text{the}\; (f \; x)) \cap \text{locs-of} \; r \; (\text{the}\; (f \; r)) = \{\} \)
      proof
        fix \( x \) assume xdom: \( x \in dom \; f - \{r\} \)
        then have xnotr: \( x \neq r \) by blast
        have xinf: \( x \in dom \; f \) using xdom by simp
        from disj show \( \text{locs-of} \; x \; (\text{the}\; (f \; x)) \cap \text{locs-of} \; r \; (\text{the}\; (f \; r)) = \{\} \)
          unfolding Disjoint-def disjnot-def Locs-of-def
          by (auto simp: xdom xnotr xinf rinf)
      qed
      qed
    proof
      also have \( \ldots = \text{locs} \; f \; - \; \text{locs-of} \; r \; (\text{the}\; (f \; r)) \) by (simp add: locs-def nat1f)
      finally show \( \exists \text{thesis by simp add: locs-def nat1f nat1-ar} \)
      qed

qed

lemma F1-inv-empty: F1-inv empty

unfolding F1-inv-def Disjoint-def sep-def nat1-map-def
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

by auto

lemma dom-ar-F1-inv:
  assumes inv: F1-inv f1
  shows F1-inv (\{l\} -a 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\} -a f1) using nat1-mapf1 by (rule dom-ar-nat1-map)
      show finite-conc: finite (dom (\{l\} -a f1)) using finitef1 by (rule dom-ar-finite)
      show sep (\{l\} -a f1) using sepf1 by (rule dom-ar-sep)
      show Disjoint (\{l\} -a f1) using disjf1 by (rule dom-ar-Disjoint)
      qed
    qed
  qed

lemma dom-ar-locs:
  assumes finite(dom f)
  and natf: nat1-map f
  and disj: Disjoint f
  and lindom: l \in dom f
  shows locs (\{l\} -a f) = (locs f) - locs-of l (the (f l))
proof -
  have locs (\{l\} -a f) = (\bigcup s \in dom (\{l\} -a f). locs-of s (the (\{l\} -a f) s))
  proof
    have nat1-map (\{l\} -a f) using natf by (rule dom-ar-nat1-map)
    thus ?thesis unfolding locs-def by simp
    qed
  also have ... = (\bigcup s \in dom (\{l\} -a f). locs-of s (the (f s)))
  by (simp add: f-in-dom-ar-the-subsume)
  also have ... = (\bigcup s \in dom f - \{l\}. locs-of s (the (f s)))
  by (simp add: l-dom-dom-ar)
  also have ... = (\bigcup s \in dom f. locs-of s (the ((f) s))) - locs-of l (the(f l))
  proof (rule UN-minus)
    show \forall s \in dom f - \{l\}. locs-of s (the (f s)) \cap locs-of l (the(f l)) = {} 
    proof
      fix s assume sdom: s \in dom f - \{l\}
      then have snotl: s \neq l by blast
      have sinf: s \in dom f using sdom by simp
      from disj show locs-of s (the (f s)) \cap 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 natf)
  qed
lemma nat1-map-upd: nat1-map f \To nat1 y \To nat1-map ( f(x \To y))
  unfolding nat1-def nat1-map-def by auto

lemma finite-map-upd: finite (dom f) \To finite (dom (f x \To y))
  by auto

lemma min-or: min x y = x \lor min x y = y by (metis min-def)

lemma sumsize2-mapupd: finite (dom f) \To x \notin dom f \To f = empty \To sum-size (f x \To y) =
  (sum-size f) + y
  unfolding sum-size-def apply simp
  by (smt setsum-cong2)

lemma setsum-mapupd: finite (dom fa) \To e \notin dom fa \To fa = empty \To \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 \To finite (dom f) \To g>0 \To sum-size (f x \To y) > 0
  unfolding sum-size-def
  by simp

lemma sum-size-singleton: sum-size [x \To y] = y
  unfolding sum-size-def
  by simp

lemma setsum-dagger: dom f \cap dom g = {} \To finite (dom f) \To \sum x\in dom f. the ((f \dagger 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) \To f \neq empty \To x \notin dom f \To sum-size (f \dagger [x \To 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)
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

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)
applysimp
apply simp
apply simp
apply (simp add: setsum-dagger)
apply subst l-dagger-commute
apply simp
by auto

lemma dagger-min: finite (dom f) ⇒ finite (dom g) ⇒ f ≠ empty ⇒ g ≠ empty ⇒ dom f ∩ dom g = {} ⇒ 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 simp-all
by (metis (mono-tags) Min-in domIff emptyE less-imp-le min-max.inf-absorb2 min-max.le-iff 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: \( f_2 = \emptyset \Rightarrow \text{min-loc}\ (f_1 \cup m f_2) = \text{min-loc}\ f_1 \)
by (metis Int-empty-right equals0D l-map-non-empty-dom-conv l-munion-apply)

lemma munion-min-loc-emptyf1: \( f_1 = \emptyset \Rightarrow \text{min-loc}\ (f_1 \cup m f_2) = \text{min-loc}\ f_2 \)
by (metis (full-types) domIff dom-eq-empty-conv inf-bot-left l-dagger-apply munion-def)

lemma dagger-min-loc-nonempty: \( \text{dom } f_1 \cap \text{dom } f_2 = \{ \} \Rightarrow \text{finite } (\text{dom } f_1) \Rightarrow \text{finite } (\text{dom } f_2) \Rightarrow \text{finite } (\text{dom } f_1 \cup \text{dom } f_2) \)
unfolding min-loc-def apply (simp add: dagger-notemptyL)
by (metis Min.Un)

lemma dagger-min-loc-emptyf2: \( f_2 = \emptyset \Rightarrow \text{min-loc}\ (f_1 \dagger f_2) = \text{min-loc}\ f_1 \)
by (metis dom-eq-empty-conv empty-iff l-dagger-apply)

lemma dagger-min-loc-emptyf1: \( f_1 = \emptyset \Rightarrow \text{min-loc}\ (f_1 \dagger f_2) = \text{min-loc}\ f_2 \)
by (metis (full-types) domIff l-dagger-apply)

lemma min-loc-singleton: \( \text{min-loc } \{ x \mapsto y \} = x \)
unfolding min-loc-def
by simp

lemma min-loc-dagger: \( \text{finite } (\text{dom } f) \Rightarrow \text{finite } (\text{dom } g) \Rightarrow f \neq \emptyset \Rightarrow g \neq \emptyset \Rightarrow \text{min-loc } (f \dagger g) = \text{min-loc } f \)
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)))
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

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: natIf)
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} -s f) ∪ m [l + s ↦→ the(f l) - s])
proof -
from inv show ?thesis
proof
assume disjf1: Disjoint f
and sepf1: sep f

end

theory HEAP1ProofsIJW
imports HEAP1LemmasIJW
begin
and \textit{nat1-map} f:
\textit{nat1-map} f

and \textit{finite} fl:
\textit{finite} (dom f)

have disjoint-dom: \(l+s \not\in \text{dom} \{l\} \ominus f\)
proof (rule l-dom-ar-not-in-dom)
show \(l+s \not\in \text{dom} f\)
proof (rule l-plus-s-not-in-f)
show \(\text{F1-inv} \ f \ \text{and} \ l \in \text{dom} \ f \ \text{and} \ s < \text{the} \ (f \ l) \ \text{and} \ \text{nat1} \ s\)
by (simp-all del: nat1-def add: inv nat1s l-in-dom f-bigger-s)
qed
qed

have \(\text{noteqls} : \forall \cdot x \in \text{dom} \ f. \ x + (\text{the} \ (f \ x)) \neq l + s\)
proof
fix \(x\)
assume \(x\in\text{dom} \ f\)
show \(x + (\text{the} \ (f \ x)) \neq l + s\)
proof (cases \(x = l\))
assume \(x = l\)
then show \(\text{?thesis}\) using \(f-bigger-s\) by simp
next
assume \(x \neq l\)
from \(\text{disjfl}\) have locs-of \(x \ (\text{the} \ (f \ x)) \cap \text{locs-of} \ l \ (\text{the} \ (f \ l)) = \{\}\)
unfolding Disjoint-def disjoint-set Locs-of-def
by (auto simp: x-in-dom xnotl l-in-dom)
moreover have \(l+s-1 \in \text{locs-of} \ l \ (\text{the} \ (f \ l))\)
by (metis \(f-bigger-s\) nat1s top-locs-of)
moreover have \(x + (\text{the} \ (f \ x)) - 1 \in \text{locs-of} \ x \ (\text{the} \ (f \ x))\)
by (metis \(\text{nat1-map-def} \ \text{nat1-map} fl \ \text{top-locs-of} \ x\in\text{dom}\))
ultimately have \(x + (\text{the} \ (f \ x)) - 1 \neq l+s-1\) by auto
thus \(\text{?thesis}\) by simp
qed
qed

next
from disjoint-dom show \(\text{finite} \ (\text{dom} \ (\{l\} \ominus f) \cup \{l+s \mapsto \text{the} \ (f \ l) - s\})\)
proof (rule unionm-singleton-finite)
  show finite \((\{l\} \ominus f)\) using finitef1 by (rule dom-ar-finite)
qed
next

from disjoint-dom show \(\text{sep} \ (\{l\} \ominus f)\)
proof (rule unionm-singleton-sep)
  show sep \((\{l\} \ominus f)\) using sepf1 by (rule dom-ar-sep)
next
show \(\forall \cdot la \in \text{dom} \ (\{l\} \ominus f). \ la + (\text{the} \ ((\{l\} \ominus f) \ la) \not\in \text{dom} \ (l+s \mapsto \text{the} \ (f \ l) - s)\)
by (metis dom-eq-singleton-conv f-in-dom-ar-subsume f-in-dom-ar-the-subsume noteqls singletonE)
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

next
show \( l + s + (the (f l) - s) \notin dom \(\{l\} -\omega f) \)
proof -
  have myfact: \( l + the (f l) \notin dom(f) \) using l-in-dom sepf1 sep-def by auto
  have \( l + the (f l) \notin dom(\{l\} -\omega f) \) 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

have disjoint \((\text{locs-of} (l + s) (the (f l) - s)) \cap (\text{locs} (\{l\} -\omega f))\)
proof -
  have \((\text{locs-of} (l + s) (the (f l) - s)) \subseteq \text{locs-of} l (the (f l))\)
  sorry
  moreover have \((\text{locs-of} l (the (f l)) \cap (\text{locs} (\{l\} -\omega f))) = \{\}\)
  sorry
ultimately show \( ?thesis \)
  by (smt Int-absorb1 Int-assoc Int-commute Int-empty-left disjoint-def)
qed

show disjoint \((\text{locs-of} (l + s) (the (f l) - s)) \cap (\text{locs} (\{l\} -\omega f))\)
proof -
  have \((\text{locs-of} (l + s) (the (f l) - s)) \subseteq \text{locs-of} l (the (f l))\)
  by (rule locs-of-subset,simp add: f-bigger-s)
  moreover have \((\text{locs-of} l (the (f l)) \cap (\text{locs} (\{l\} -\omega f))) = \{\}\)
proof
  subst l-locs-of-dom-ar
  show nat1-map f and Disjoint f and \( l \in dom f \)
    by (simp-all add: l-in-dom nat1-mapf1 disj1)
next
show \((\text{locs-of} l (the (f l)) \cap (\text{locs} f - \text{locs-of} l (the (f l)))) = \{\}\)
  by simp
qed
ultimately show \( ?thesis \)
  by (smt Int-absorb1 Int-assoc Int-commute Int-empty-left disjoint-def)
qed
qed
qed
qed

171
lemma (in level1-new) new1-post-feaseq:
 assumes pre-eq: ∃· l ∈ dom f1. the (f1 l) = s1
 shows ∃· r f1new. new1-post-eq f1 s1 f1new r ∧ F1-inv f1new
proof -
  from pre-eq obtain l where ind: l ∈ dom f1 and preinstance: the (f1 l) = s1 ..
  obtain f1new where f1wit: f1new = {l} -o f1 by auto
  from ind and preinstance and f1wit have l ∈ dom f1 ∧ the (f1 l) = s1 ∧ f1new = {l} -o 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: ∃· l ∈ dom f1. the (f1 l) > s1
 shows ∃· r f1new. new1-post-gr f1 s1 f1new r ∧ F1-inv f1new
proof -
  from pre-gr obtain l where ind: l ∈ dom f1 and preinstance: the (f1 l) > s1 ..
  obtain f1new where f1wit: f1new = ({l} -o f1) ∪ m [l + s1 ↦ the(f1 l) - s1] by auto
  from ind and preinstance and f1wit have l ∈ dom f1 ∧ the (f1 l) > s1 ∧ f1new = ({l} -o f1) ∪ m [l + s1 ↦ the(f1 l) - s1]
    by simp
  moreover have F1-inv f1new
  proof -
    have F1-inv ((({l} -o f1) ∪ m [l + s1 ↦ the(f1 l) - 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)
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

lemma (in level1-dispose) disjoint-above-below[simp]:
  dom(dispose1-above f1 d1 s1) \cap dom(dispose1-below f1 d1) = {}

unfolding dispose1-above-def dispose1-below-def
proof (rule l-dom-r-disjoint-weakening)
  show \{x \in dom f1. x = d1 + s1\} \cap \{x \in dom f1. x + the (f1 x) = d1\} = {}
    proof (cases \{x \in dom f1. x = d1 + s1\} = {})
      assume \{x \in dom f1. x = d1 + s1\} = {}
      then show \{x \in dom f1. x = d1 + s1\} \cap \{x \in dom f1. x + the (f1 x) = d1\} = {}
        by auto
    next
      assume \*: \{x \in dom f1. x = d1 + s1\} \neq {}
      show \{x \in dom f1. x = d1 + s1\} \cap \{x \in dom f1. x + the (f1 x) = d1\} = {}
        proof (cases \{x \in dom f1. x = d1 + s1\} \neq {})
          assume \{x \in dom f1. x = d1 + s1\} \neq {}
          then show \{x \in dom f1. x = d1 + s1\} \cap \{x \in dom f1. x + the (f1 x) = d1\} = {}
            proof (rule ccontr)
              assume nonempty: \{x \in dom f1. x = d1 + s1\} \cap \{x \in dom f1. x + the (f1 x) = d1\} \neq {} from \* \* obtain x where xinter: x \in \{x \in dom f1. x = d1 + s1\} \cap \{x \in dom f1. x + the (f1 x) = d1\} = 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
      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 \notin dom (dispose1-above f1 d1 s1)
unfolding dispose1-above-def
proof (subst l-dom-r-subseteq)
  show \{x \in dom f1. x = d1 + s1\} \subseteq dom f1
    by auto
  next
    show d1 \notin \{x \in 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 \notin \text{dom} (\text{dispose1-below } f1 \ d1)

unfolding dispose1-below-def

proof (subst l-dom-r-subseteq)
  show \{x \in \text{dom } f1. x + \text{the } (f1 \ x) = d1\} \subseteq \text{dom } f1
    by auto
  next
  show d1 \notin \{x \in \text{dom } f1 . x + \text{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 \notin \text{dom} (\text{dispose1-above } f1 \ d1 \cup_m \text{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: \text{dom} (\text{dispose1-ext } f1 \ d1 \ s1) = \text{dom} (\text{dispose1-above } f1 \ d1 \ s1) \cup \text{dom} (\text{dispose1-below } f1 \ d1) \cup \{d1\}

proof -
  have \text{dom} (\text{dispose1-ext } f1 \ d1 \ s1) = \text{dom} (\text{dispose1-above } f1 \ d1 \ s1) \cup \text{dom} (\text{dispose1-below } f1 \ d1) \cup \{d1\}
    unfolding dispose1-ext-def
  by (rule l-munion-dom, simp add: d1-not-above-below)
  also have ... = \text{dom} (\text{dispose1-above } f1 \ d1 \ s1 \cup \text{dispose1-below } f1 \ d1) \cup \{d1\}
    unfolding munion-def by simp
  finally show ?thesis by (simp add: l-dagger-dom)

qed

lemma (in level1-dispose) dispose1-ext-notempty: \text{dispose1-ext } f1 \ d1 \ s1 \neq \text{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: \text{dom } (\text{dispose1-ext } f1 \ d1 \ s1) \neq \{\}
  by (metis Un-insert-right dispose1-ext-union insert-not-empty)

lemma (in level1-dispose) d1notinf1: d1 \notin \text{dom } f1

proof -
  have \text{dom } f1 \subseteq \text{locs } f1
    proof (rule domf-in-locs)
      show \text{nat1-map } f1 by (metis invF1-nat1-map-weaken l1-invariant-def)
    qed
  moreover have d1 \in \text{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: \text{min-loc } (\text{dispose1-ext } f1 \ d1 \ s1) = \text{Min} (\text{dom } (\text{dispose1-above } f1 \ d1 \ s1)) \cup (\text{dom } (\text{dispose1-below } f1 \ d1)) \cup \{d1\}

proof -
  have \text{min-loc } (\text{dispose1-ext } f1 \ d1 \ s1) = \text{min-loc} (\text{dispose1-above } f1 \ d1 \ s1 \cup \text{dispose1-below } f1 \ d1 \cup_m [d1 \mapsto s1])
unfolding \texttt{dispose1-ext-def} by simp
also have $... = \min (\dom (\texttt{dispose1-above } f1 \ d1 \ s1) \cup m \ \texttt{dispose1-below } f1 \ d1 \cup m [d1 \mapsto s1])$
unfolding \texttt{min-loc-def}
by (fold \texttt{dispose1-ext-def}, simp add: \texttt{dispose1-ext-notempty})
also have $... = \min ((\dom (\texttt{dispose1-above } f1 \ d1 \ s1)) \cup (\dom (\texttt{dispose1-below } f1 \ d1)) \cup \{d1\})$
by (fold \texttt{dispose1-ext-def}, simp add: \texttt{dispose1-ext-union})
finally show ?thesis by simp
qed

lemma \texttt{above-dom}:
assumes \texttt{above-notempty}: $(\texttt{dispose1-above } f1 \ d1 \ s1) \neq \emptyset$
shows \texttt{dom (dispose1-above } f1 \ d1 \ s1) = \{d1 + s1\}
proof -
have \texttt{dispose1-above } f1 \ d1 \ s1 = \{x \in \dom f1 . x = d1 + s1 \} @ f1
by (metis \texttt{dispose1-above-def})
then have \{x \in \dom f1 . x = d1 + s1 \} @ \emptyset
by (metis \texttt{above-notempty l-dom-r-nothing})
moreover have \texttt{dom (dispose1-above } f1 \ d1 \ s1) = \{d1 + s1\}
unfolding \texttt{dispose1-above-def}
proof (subst \texttt{l-dom-r-iff})
show \{x \in \dom f1 . x = d1 + s1 \} @ \dom f1 = \{d1 + s1\}
by (metis \texttt{Collect-conj-eq Collect-conv-if Collect-mem-eq calculation inf-commute singleton-conv})
qed
thus ?thesis .
qed

lemma \texttt{above-min-loc}:
assumes \texttt{above-notempty}: $(\texttt{dispose1-above } f1 \ d1 \ s1) \neq \emptyset$
shows \texttt{min-loc (dispose1-above } f1 \ d1 \ s1) = d1 + s1
unfolding \texttt{min-loc-def}
by (metis \texttt{Min-singleton assms above-dom})

lemma \texttt{above-d1s1-in-f1}:
assumes \texttt{above-notempty}: $(\texttt{ dispose1-above } f1 \ d1 \ s1) \neq \emptyset$
shows \texttt{d1+s1 \in \dom f1}
proof -
have \texttt{dom (dispose1-above } f1 \ d1 \ s1) @ \dom (f1)
unfolding \texttt{dispose1-above-def} by (simp add: \texttt{l-dom-r-dom-subseteq})
moreover have \{d1+s1\} @ \dom f1 by (metis \texttt{above-dom assms calculation})
ultimately show ?thesis by auto
qed

lemma \texttt{above-sumsize}:
assumes \texttt{above-notempty}: $(\texttt{dispose1-above } f1 \ d1 \ s1) \neq \emptyset$
shows \texttt{sum-size (dispose1-above } f1 \ d1 \ s1) = \texttt{the (f1 (d1 + s1))}
unfolding \texttt{sum-size-def}
apply (simp add: \texttt{above-notempty})
apply (subst \texttt{above-dom})
apply (rule \texttt{above-notempty})
unfolding \texttt{dispose1-above-def}
apply (subgoal-tac \{x . x = d1 + s1 \} @ \dom f1 = \{d1+s1\})
apply (simp)
apply (subst \texttt{f-in-dom-r-apply-elem})
apply simp-all
by (metis \texttt{Collect-conj-eq Collect-mem-eq Int-empty-left}

APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)
lemma (in level1-dispose) d1-above:
∀ x ∈ dom (dispose1-above f1 d1 s1), x > d1
by (metis (mono_tags) above-dom d1-not-dispose-above
l-map-non-empty-has-elem-cone less-le not-add-less1 not-less singletonE)

lemma (in level1-dispose) min-below-empty:
assumes below-empty: dom (dispose1-below f1 d1) = {}
shows min-loc (dispose1-ext f1 d1 s1) = d1
proof(cases dom (dispose1-above f1 d1 s1) = {})
  assume dom (dispose1-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-above f1 d1 s1) ≠ {}
  have min-loc (dispose1-ext f1 d1 s1) = Min (dom (dispose1-above f1 d1 s1) ∪ {d1})
    by (simp add: below-empty min-loc-unfold)
  also have ... = min (Min (dom (dispose1-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-below f1 d1))) d1 by simp
  finally show ?thesis by (metis Min-singleton above-dom above-notempty
    l-map-non-empty-dom-cone le-add1 min-absorb1 min-max inf-commute)
qed

lemma dom-ar-disjoint: (dom f) ∩ (Y) = {} ⇒ (dom (X - f)) ∩ Y = {}
by (metis Diff-Int-distrib2 empty-Diff l-dom-dom-ar)

lemma (in level1-dispose) min-below-notempty:
assumes below-notempty: dom (dispose1-below f1 d1) ≠ {}
shows min-loc (dispose1-ext f1 d1 s1) ∈ dom (dispose1-below f1 d1)
proof
  have Min (dom (dispose1-above f1 d1 s1) ∪ dom (dispose1-below f1 d1) ∪ {d1})
    = Min (dom (dispose1-below f1 d1) ∪ (dom (dispose1-above f1 d1 s1) ∪ {d1}))
    by (metis Un-insert-left min-loc-unfold sup-bot-left sup-commute)
  also have ... = min (Min (dom (dispose1-below f1 d1))) d1 by simp
  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-below f1 d1))) d1
  proof(cases dom (dispose1-above f1 d1 s1) = {})
    assume dom (dispose1-above f1 d1 s1) = {} thus ?thesis by simp
  next
    assume dom (dispose1-above f1 d1 s1) ≠ {} then
    have Min (dom (dispose1-above f1 d1 s1) ∪ {d1}) = Min ({d1+s1} ∪ {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-below f1 d1))
  proof -
have ∗: ∃· 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 + (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 ...
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)) -o 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)) -o 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)) -o 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)) -o 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)) -o f1))
        by (metis dom-empty dom-fun-upd finite finite-insert option.distinct(1))
      qed
      next
      show VDM-F1-inv
        ((dom (dispose1-below f1 d1)) ∪ dom (dispose1-above f1 d1 s1)) -o 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
        show ?thesis
          proof (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)
            next
            show ?thesis
        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)) -o 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)) -o 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)) -o 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)) -o f1))
              by (metis dom-empty dom-fun-upd finite finite-insert option.distinct(1))
            qed
            next
            show VDM-F1-inv
              ((dom (dispose1-below f1 d1)) ∪ dom (dispose1-above f1 d1 s1)) -o 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
              show ?thesis
                proof (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)
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

then show \(\text{thesis}\)
proof (cases dispose1-above \(f_1\) \(d_1\) \(s_1\) = empty)
assume above-empty: dispose1-above \(f_1\) \(d_1\) \(s_1\) = empty
then show \(\text{thesis}\)
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 \((f_1 \cup m \[d_1 \mapsto s_1\])\)
proof
show sep \((f_1 \cup m \[d_1 \mapsto s_1\])\)
proof (rule unionm-singleton-sep)
show sep \(f_1\) by (rule sepf1)
next
show \(\forall l \in \text{dom }f_1. l + \text{the}(f_1 l) \notin \text{dom }[d_1 \mapsto s_1]\)
proof
fix \(l\)
assume \(l \in \text{dom }f_1\)
have \(l + \text{the}(f_1 l) \neq d_1\)
proof -
have dispose1-below \(f_1\) \(d_1\) \(s_1\) = \(\{ x \in \text{dom }f_1. x + \text{the}(f_1 x) = d_1 \}\) \(\triangleq f_1\)
unfolding dispose1-below-def by simp
then have \(\{ x \in \text{dom }f_1. x + \text{the}(f_1 x) = d_1 \}\) = \(\{\}\)
by (smt IntI below-empty dom-def dom-eq-empty-conv
empty-Collect-eq l-dom-r-iff mem-Collect-eq)
thus \(\text{thesis}\) by (smt \(l \in \text{dom }f_1\) empty-Collect-eq)
qed
then show \(l + \text{the}(f_1 l) \notin \text{dom }[d_1 \mapsto s_1]\)
by simp
qed
next
show \(d_1 + s_1 \notin \text{dom }f_1\)
proof -
have dispose1-above \(f_1\) \(d_1\) \(s_1\) = \(\{ x \in \text{dom }f_1. x = d_1 + s_1 \}\) \(\triangleq f_1\)
unfolding dispose1-above-def by simp
then have \(\{ x \in \text{dom }f_1. x = d_1 + s_1 \}\) = \(\{\}\)
by (smt Collect-empty-ck above-empty disjoint-iff-not-equal
dom-def l-dom-r-iff mem-Collect-eq)
thus \(\text{thesis}\) by (smt empty-Collect-eq)
qed
next
show \(d_1 \notin \text{dom }f_1\) by (rule d1notinf1)
next
show \(\text{nat1 } s_1\) by (simp only: II-input-notempty-def)
next
show Disjoint \((f_1 \cup m \[d_1 \mapsto s_1\])\)
proof (rule unionm-singleton-Disjoint)
show Disjoint \(f_1\) by (rule disjf1)
next
show \(\text{nat1 } \text{map } f_1\) by (rule nat1-mapf1)
next
show \(d_1 \notin \text{dom }f_1\) by (rule d1notinf1)
next
show \(\text{nat1 } s_1\) by (rule II-input-notempty-def)
next
from II-dispose1-precondition-def show disjoint \((\text{locs-of }d_1 \ s_1)\) \((\text{locs }f_1)\)
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)) union dom (dispose1-above f1 d1 s1)) = {d1 + s1}
by (simp add: below-empty l-munion-empty-rhs l-munion-empty-lhs l-dom-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 minbelow-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-empty-lhs d1notinf1
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 \cup m [d1 \mapsto the (f1 (d1 + s1)) + s1])
  proof
    show sep (\{d1 + s1\} -\& f1 \cup m [d1 \mapsto the (f1 (d1 + s1)) + s1])
    proof (rule unionm-singleton-sep)
      show sep (\{d1 + s1\} -\& f1) using sep f1 by (rule dom-ar-sep)
    next
    have d1 \notin dom (\{d1 + s1\} -\& f1)
    proof -
      have d1 \notin dom f1 by (rule d1notinf1)
      thus ?thesis by (metis l-dom-ar-notin-dom-or)
    qed
    next
    show \forall l \in dom (\{d1 + s1\} -\& f1).
      \ l + the (\{d1 + s1\} -\& f1) l
      \notin dom [d1 \mapsto the (f1 (d1 + s1)) + s1]
    proof
      fix l assume lindom: l \in dom (\{d1 + s1\} -\& f1)
      then have linf: l \in dom f1 by (metis l-dom-ar-notin-dom-or)
      have l + the (f1 l) ≠ d1
      proof -
        have dispose1-below f1 d1 = \{ x \in dom f1 . x + the(f1 x) = d1 \} -\& f1
        unfolding dispose1-below-def by simp
        then have \{ x \in 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 \notin dom [d1 \mapsto the (f1 (d1 + s1)) + s1] by auto
      qed
      next
      show d1 + (the (f1 (d1 + s1)) + s1) \notin dom (\{d1 + s1\} -\& f1)
      proof -
        have sep f1 by (rule sep f1)
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

then have $\forall \cdot \in \text{dom} \ f1. \ l + \text{the} \ (f1 \ l) \notin \text{dom} \ f1$
  using sep-def by auto
  moreover have $\{d1 + s1\} \in \text{dom} \ f1$ using above-notempty by (rule above-d1s1-in-f1)
  moreover have $(d1 + s1) + \text{the} \ (f1 \ (d1 + s1)) \notin \text{dom} \ f1$
    by (metis calculation(1) calculation(2))
  ultimately show $d1 + (\text{the} \ ((f1 \ (d1 + s1)) + s1)) \notin \text{dom} \ (\{d1 + s1\} - \circ \ f1)$
    by (smt f-in-dom-ar-subsume)
  qed
next
show $\text{nat}1 \ (\text{the} \ ((f1 \ (d1 + s1)) + s1))$
  by (simp, rule disjF1, metis l1-input-notempty-def nat1-def)
  qed
next
show Disjoint ($\{d1 + s1\} - \circ \ f1 \cup m : d1 \mapsto \text{the} \ (f1 \ (d1 + s1)) + s1)$
proof (rule unionm-singleton-Disjoint)
  show Disjoint ($\{d1 + s1\} - \circ \ f1$) using disjF1 by (rule dom-ar-Disjoint)
next
show $d1 \notin \text{dom} \ (\{d1 + s1\} - \circ \ f1)$ using d1notinF1 by (simp add: l-dom-ar-notin-dom-or)
next
show nat1-map ($\{d1 + s1\} - \circ \ f1$) using nat1-mapF1 by (rule dom-ar-nat1-map)
next
show $\text{nat}1 \ (\text{the} \ ((f1 \ (d1 + s1)) + s1))$
  by (metis (mono-tags) add-eq-if l1-input-notempty-def nat1-def zero-less-Suc)
next
show disjoint (locs-of $d1 \ (\text{the} \ ((f1 \ (d1 + s1)) + s1)) \ (\text{locs} \ (\{d1 + s1\} - \circ \ f1))$
proof -
  from l1-dispose1-precondition-def have disjoint (locs-of $d1 \ s1$) (locs $f1$)
    by (simp add: dispose1-pre-def)
  moreover have (locs ($\{d1 + s1\} - \circ \ f1$)) = 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 $d1 \ (\text{the} \ ((f1 \ (d1 + s1)) + s1))$
    =
      locs-of $d1 \ s1$
      $\cup$
      locs-of ($d1 + s1$) (the ($f1 \ (d1 + s1)\))
  proof (subst add-commute, rule locs-of-sum-range)
  show $\text{nat}1 \ (\text{the} \ ((f1 \ (d1 + s1))))$
    by (metis (full-types) above-d1s1-in-f1 above-notempty nat1-map-def nat1-mapF1)
next
show $\text{nat}1 \ s1$ by (metis l1-input-notempty-def)
  qed
  ultimately show ?thesis unfolding disjoint-def by auto
  qed
  qed
  qed
  qed
next
assume below-notempty: dispose1-below $f1 \ d1 \neq \text{empty}$
from below-notempty have $\exists \cdot x. \ x \in \text{dom} \ f1 \land x + \text{the} \ (f1 \ x) = d1$
proof -
  have dispose1-below $f1 \ d1 \neq \text{empty}$ by (rule below-notempty)
  then have $\{ x \in \text{dom} \ f1 . \ x + \text{the}(f1 \ x) = d1 \} \neq \{\}$
    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)
    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 (no_types) 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 (no_types) l-in-cone 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
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

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 (rule unionm-singleton-sep)
  show sep ({below} -< a f₁) using sep₁ by (rule dom-ar-sep)
next
  show below ∉ dom ({below} -< a f₁) by (metis f-in-dom-ar-notelem)
next
  show ∀ l ∈ dom ({below} -< a f₁).
    l + the ((({below} -< a f₁) l) ∉ dom [below ↦ the (f₁ below) + s₁])
proof
  fix l assume lin-restr-dom: l ∈ dom ({below} -< a f₁)
  have l ∈ dom f₁ using lin-restr-dom by (metis l-dom-ar-not-in-dom)
  have l + the ((({below} -< a f₁) l) ∉ dom [below ↦ the (f₁ below) + s₁])
    by (metis l ∈ dom f₁ belowinf₁ f-in-dom-ar-apply-subsume
        lin-restr-dom sep-def sep₁)
  thus l + the ((({below} -< a 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} -< a f₁)
proof -
  have below + (the (f₁ below)) = d₁
    by (metis belowplusf₁below)
  then have d₁ + s₁ ∉ dom ({below} -< a 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-Collact-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
      domIf 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} -< a f₁ ∪ m [below ↦ the (f₁ below) + s₁])
proof (rule unionm-singleton-Disjoint)
  show below ∉ dom ({below} -< a f₁)

by (simp add: below-in-dom dom-below l-dom-ar-notin-dom-or)
next
show Disjoint ({below} -\langle f1 \rangle) using disjf1 by (rule dom-ar-Disjoint)
next
show nat1-map ({below} -\langle f1 \rangle) 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} -\langle f1 \rangle))
proof -
  from ll-dispose1-precondition-def have disjoint (locs-of d1 s1) (locs f1)
  by (simp add: dispose1-pre-def)
  moreover have (locs ({below} -\langle f1 \rangle)) = 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))
    \cup 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 ll-input-notempty-def)
    qed
  qed
  ultimately show \{?thesis unfolding disjoint-def by auto\}
qued
qed

next
assume above-notempty: dispose1-above f1 d1 s1 \neq \text{Map.empty}
have above-below-shape: (dom (dispose1-below f1 d1) \cup dom (dispose1-above f1 d1 s1))
  = \{\text{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 \cup 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 \cup m dispose1-below f1 d1)) and
          finite (dom [d1 \mapsto s1])
          by (simp-all add: finite-dispose1-above finite-dispose1-below k-finite-munion)
      next
      show dispose1-above f1 d1 s1 \cup m dispose1-below f1 d1 \neq empty
        and [d1 \mapsto s1] \neq empty
        by (auto simp: munion-notempty-right below-notempty)
      next
      from d1-not-above-below show dom (dispose1-above f1 d1 s1 \cup m dispose1-below f1 d1) \cap dom
        [d1 \mapsto s1] = \{\}
    qed
  qed

next

APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

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

do proof
proof (simp add: sum-size-shape min-loc-shape above-below-shape) show VDM-F1-inv ({below, d1 + s1} -α f1 ∪ m [below ↦ the (f1 (d1 + s1)) + the (f1 below) + s1])

proof
show sep ({below, d1 + s1} -α f1 ∪ m [below ↦ the (f1 (d1 + s1)) + the (f1 below) + s1])
proof (rule unionm-singleton-sep)

show sep ({below, d1 + s1} -α f1) using sep f1 by (rule dom-ar-sep)
next
show below ≠ dom ({below, d1 + s1} -α f1)
by (metis insertII l-dom-ar-notin-dom-or)
next
show ∀ l ∈ dom ({below, d1 + s1} -α f1).
  l + the ({below, d1 + s1} -α f1) l ≠ dom [below ↦ the (f1 (d1 + s1)) + the (f1 below) + s1]
proof
fix l assume lin-restr-dom: l ∈ dom ({below, d1 + s1} -α f1)
have l ∈ dom f1 using lin-restr-dom by (metis l-dom-ar-notin-dom)

have l + the ({below, d1 + s1} -α f1) l ≠ below
by (metis l ∈ dom f1; belowinf1 f-in-dom-ar-apply-subsame
lin-restr-dom sep-def sep f1)
thus l + the ({below, d1+s1} -α 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} -α f1)
proof

have below + (the (f1 below)) = d1
by (metis belowplusf1below)

then have (d1 +s1) + (the (f1 (d1 + s1))) ≠ dom ({below,d1+s1} -α f1)
by (metis above-dis1-in-f1 above-notempty l-dom-ar-notin-dom-or sep-def sep f1)
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} -α f1 ∪ m [below ↦ the (f1 (d1 + s1)) + the (f1 below) + s1])
proof (rule unionm-singleton-Disjoint)

show below ≠ dom ({below, d1 + s1} -α f1)
by (metis insertII l-dom-ar-notin-dom-or)
next
show Disjoint ({below, d1 + s1} -α f1) using disj f1 by (rule dom-ar-Disjoint)
next
show nat1-map ({below, d1 + s1} -α f1) using nat1-map f1 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-loc)
  show finite (dom ((d1 + s1) -\& f1)) by (metis dom-ar-finite finitef1)
next
  show natl-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 \in 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 \notin (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-loc, 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))) \cup
  (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) ll-input-notempty-def nat1-def trans-less-add2)
qed
also have ... = (locs-of below (the (f1 below))) \cup
  (locs-of d1 (the (f1 (d1 + s1)) + s1))
  by (metis belowplusf1below)
also have ... = (locs-of below (the (f1 below))) \cup
  locs-of d1 (s1 + (the (f1 (d1 + s1))))
  by (metis nat-add-commute)
also have ... = (locs-of below (the (f1 below)))
  \cup locs-of d1 s1
  \cup locs-of (d1+s1) (the (f1 (d1 + s1)))
proof (subst locs-of-sum-range)
  show nat1 s1 by (metis ll-input-notempty-def)
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

next
  show \( \text{nat1 \ (the \ (f1 \ (d1 + s1)))} \)
  by (metis above-d1s1-in-f1 above-notempty nat1-map-def nat1-mapf1)
qed (metis sup-commute sup-left-commute)

finally have **: \( \text{locs-of below \ (the \ (f1 \ (d1 + s1))) + the \ (f1 \ below) + s1} \)
  \( = (\text{locs-of below \ (the \ (f1 \ below)))} \)
  \( \cup \text{locs-of \ d1 \ s1} \)
  \( \cup \text{locs-of \ (d1+s1) \ (the \ (f1 \ (d1 + s1)))} \)
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) \cup dom (dispose1-above f1 d1 s1))
    -\(\text{f1} \cup m\)
      \(\text{[min-loc} \ (\text{dispose1-ext \ f1 \ d1 \ s1}) \rightarrow \text{sum-size} \ (\text{dispose1-ext \ f1 \ d1 \ s1})]\)
    by auto
    from f1wit F1-inv-dispose show \( \exists f' \cdot \text{dispose1-post2} \ f1 \ d1 \ s1 \ f' \land F1-inv \ f' \)
      using dispose1-post2-def by (metis l1-invariant-def)
  qed

end

theory HEAP1SanityIJW
importst 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''
  qed

end
proof\(\) new1-post-eq-def

\begin{align*}
&\text{assume } r -\in -\text{dom} \quad r \in \text{dom } f \\
\text{and eq-n: } &\text{the } ((f \ r) = n) \\
\text{and } f' -\text{restr: } &f' = \{r\} -\cdot a f
\end{align*}

have below-shape: dispose1-below \(\{r\} -\cdot a f\) \(r = \text{empty}\)

unfolding dispose1-below-def

proof (rule l-dom-r-nothing-empty)

\begin{align*}
&\text{from inv show } \{x \in \text{dom } \{r\} -\cdot a f. \ x + \text{the } ((\{r\} -\cdot a f) \ x) = r\} = \{\}
\end{align*}

proof

assume \(\text{sep-f}: \text{sep } f\)

have \(\{x \in \text{dom } \{r\} -\cdot a f. \ x + \text{the } ((\{r\} -\cdot a f) \ x) = r\}

\begin{align*}
&= \{x \in \text{dom } \{r\} -\cdot a f. \ x + \text{the } (f x) = r\}
&\text{by (metis \text{full-types} \text{Diff} -\cdot a f \text{in-dom-ar \text{apply} -\text{not-elem}})}
\end{align*}

also have \(\subseteq \{x \in \text{dom } (f). \ x + \text{the } (f x) = r\}

\begin{align*}
&\text{by (smt Collect-empty-eq \text{l-dom-ar-not-in-dom r-in-dom sep-def sep-f subsetI})}
\end{align*}

also have \(\{\} = \{\}

\begin{align*}
&\text{by (smt Collect-empty-eq r-in-dom sep-def sep-f)}
\end{align*}

finally show \(?\text{thesis by simp}\)

qed

have above-shape: dispose1-above \(\{r\} -\cdot a f\) \(r n = \text{empty}\)

unfolding dispose1-above-def

proof (rule l-dom-r-nothing-empty)

\begin{align*}
&\text{from inv show } \{x \in \text{dom } \{r\} -\cdot a f. \ x = r + n\} = \{\}
\end{align*}

proof

assume \(\text{sep-f}: \text{sep } f\)

have \(\{x \in \text{dom } \{r\} -\cdot a f. \ x = r + n\} \subseteq \{x \in \text{dom } (f). \ x = r + n\}

\begin{align*}
&\text{by (smt Collect-empty-eq equals0D \text{l-dom-ar-not-in-dom subsetI})}
\end{align*}

also have \(\{\} = \{\}

\begin{align*}
&\text{by (smt empty-Collect-eq eq-n r-in-dom sep-def sep-f)}
\end{align*}

finally show \(?\text{thesis by simp}\)

qed

have min-loc-shape: min-loc \((\text{dispose1-ext } (\{r\} -\cdot a f) n) = r)

unfolding dispose1-ext-def

proof

have \(\text{dispose1-above } ((\{r\} -\cdot a f) r n \cup m \text{ dispose1-below } (\{r\} -\cdot a f) r \cup m [r \mapsto n])

\begin{align*}
&= [r \mapsto n]
&\text{by (simp add: above-shape l-munion-empty-lhs below-shape)}
\end{align*}

moreover have \(\text{min-loc } [r \mapsto n] = r\)

unfolding min-loc-def

by simp

ultimately show \(\text{min-loc } (\text{dispose1-above } ((\{r\} -\cdot a f) n \cup m \text{ dispose1-below } (\{r\} -\cdot a f) r \cup m [r \mapsto n])) = r\)

by simp

qed

have sum-size-shape: sum-size \((\text{dispose1-ext } (\{r\} -\cdot a f) n) = \text{the}(f r)\)

unfolding dispose1-ext-def

proof

have \(\text{dispose1-above } ((\{r\} -\cdot a f) r n \cup m \text{ dispose1-below } (\{r\} -\cdot a f) r \cup m [r \mapsto n])

\begin{align*}
&= [r \mapsto n]
\end{align*}
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

by (simp add: above-shape l-munion-empty-lhs below-shape)
moreover have sum-size \([r \mapsto n]\) = n
unfolding sum-size-def by simp
moreover have the \((f \circ r) = n\) by (rule eq-n)
ultimately show sum-size \(\text{dispose1-above} \\{(f \circ r) \mapsto n\} = \text{the} \ (f \circ r)\)
by simp
qed

from \(d1\)-post show \(?thesis\)
proof (simp only: dispose1-equiv, unfold dispose1-post2-def)
assume \(f'' = (\text{dom} \ (\text{dispose1}-\text{below} \ f') \cup \text{dom} \ (\text{dispose1}-\text{above} \ f' \ r \ n) -\circ f' \cup m\)
[min-loc \ (\text{dispose1}-\text{ext} \ f' \ r \ n) \mapsto \text{sum-size} \ (\text{dispose1}-\text{ext} \ f' \ r \ n)\]
then have \(f'' = ((\text{dom} \ (\text{empty}) \cup \text{dom} \ (\text{empty}))) -\circ f'\)
\(\cup m \ \text{min-loc} \ (\text{dispose1}-\text{ext} \ f' \ r \ n) \mapsto \text{sum-size} \ (\text{dispose1}-\text{ext} \ f' \ r \ n)\)
by (simp add: f'-restr below-shape above-shape)
then have \(f'' = \{\} -\circ f'\)
\(\cup m \ \text{min-loc} \ (\text{dispose1}-\text{ext} \ f' \ r \ n) \mapsto \text{sum-size} \ (\text{dispose1}-\text{ext} \ f' \ r \ n)\)
by simp
also have ... = \(f'\)
\(\cup m \ \text{min-loc} \ (\text{dispose1}-\text{ext} \ f' \ r \ n) \mapsto \text{sum-size} \ (\text{dispose1}-\text{ext} \ f' \ r \ n)\)
by (metis l-dom-ar-empty-lhs)
also have ... = \(\{\} -\circ f \cup m \ [r \mapsto \text{the} \ (f \circ r)\]\)
by (simp add: min-loc-shape sum-size-shape f'-restr)
also have ... = \(\{(f \circ r) -\circ f \} \cup m \ [r \mapsto \text{the} \ (f \circ r)\]\)
by (simp add: f'-restr)
also have ... = \(\{(f \circ r) -\circ f \} \uplus [r \mapsto \text{the} \ (f \circ r)]\)
proof -
  have dom \((f \circ r) -\circ f\) \cap dom \([r \mapsto \text{the} \ (f \circ r)]\) = \{\}
  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 ... = \(\text{r-in-dom}\) by (rule antirestr-then-dagger)
finally show \(?thesis\) \.
qed

next
assume new1-post-gr \(f \circ n \circ f' \ r\)
then show \(?thesis\)
unfolding new1-post-gr-def
proof (elim conjE)
assume r-in-dom: \(r \in \text{dom} \ f\)
and gr-n: \(\text{the} \ (f \circ r) > n\)
and f'-restr: \(f' = \{r\} -\circ f \cup m \ [r + n \mapsto \text{the} \ (f \circ r) - n]\)

have disjoint-dom: \(\text{dom} \ f \cap \text{dom} \ [r + n \mapsto \text{the} \ (f \circ r) - n] = \{\}\)
proof (simp)
show \(r + n \notin \text{dom} \ f\)
proof (rule l-plus-s-not-in-f)
  show F1-inv \(f\) by (metis inv)
next
show \(r \in \text{dom} \ f\) by (rule r-in-dom)
next
show \(n < \text{the} \ (f \circ r)\) by (rule gr-n)
next
show nat\(1\) \(n\) by (rule nat\(1\)n)
\begin{verbatim}
qed

have disjoint-dom-antirestr: \(\text{dom} (\{r\} \rightarrow f) \cap \text{dom} [r + n \mapsto \text{the} (f r) \cdot n] = \{\}\)
by (metis disjoint-dom-antirestr nondisjoint-dom-antirestr)

have below-shape: dispose1-below (\{r\} \rightarrow f \cup m [r + n \mapsto \text{the} (f r) \cdot n]) \(\Rightarrow \) \(r = \text{empty}\)
unfolding dispose1-below-def
proof (rule l-dom-r-nothing-empty)
from inv show \(\{x \in \text{dom} (\{r\} \rightarrow f \cup m [r + n \mapsto \text{the} (f r) \cdot n])\).
\(x + \text{the} (\{\{r\} \rightarrow f \cup m [r + n \mapsto \text{the} (f r) \cdot n]) x = r\) = \(\{\}\)
proof
assume sep: sep \(f\)
  have (\(\{r\} \rightarrow f \cup m [r + n \mapsto \text{the} (f r) \cdot n]\)) = (\(\{r\} \rightarrow f \uparrow [r + n \mapsto \text{the} (f r) \cdot n]\))
  by (metis disjoint-dom-antirestr non_empty_def)
then have \(\{x \in \text{dom} (\{r\} \rightarrow f \cup m [r + n \mapsto \text{the} (f r) \cdot n])\).
\(x + \text{the} (\{\{r\} \rightarrow f \cup m [r + n \mapsto \text{the} (f r) \cdot n] x = r\)
= \(\{x \in \text{dom} (\{r\} \rightarrow f \uparrow [r + n \mapsto \text{the} (f r) \cdot n])\).
\(x + \text{the} (\{\{r\} \rightarrow f \uparrow [r + n \mapsto \text{the} (f r) \cdot n] x = r\)
by simp
also have \(\cdots = \{x \in \text{dom} (\{r\} \rightarrow f).\)
\(x + \text{the} (\{\{r\} \rightarrow f \cup m [r + n \mapsto \text{the} (f r) \cdot n] x = r\)
\(x + \text{the} (\{r + n \mapsto \text{the} (f r) \cdot n] x = r\)
proof (subt l-dagger-dom)
show \(\{x \in \text{dom} (\{r\} \rightarrow f) \cup m [r + n \mapsto \text{the} (f r) \cdot n].\)
\(x + \text{the} (\{\{r\} \rightarrow f \uparrow [r + n \mapsto \text{the} (f r) \cdot n] x = r\)
= \(\{x \in \text{dom} (\{r\} \rightarrow f) \cup m [r + n \mapsto \text{the} (f r) \cdot n].\)
\(x + \text{the} (\{\{r\} \rightarrow f \uparrow [r + n \mapsto \text{the} (f r) \cdot n] x = r\)
proof -
have \(\{x \in \text{dom} (\{r\} \rightarrow f), x + \text{the} (\{\{r\} \rightarrow f \uparrow [r + n \mapsto \text{the} (f r) \cdot n] x = r\)
= \(\{x \in \text{dom} (\{r\} \rightarrow f), x + \text{the} (\{\{r\} \rightarrow f \uparrow [r + n \mapsto \text{the} (f r) \cdot n] x = r\)
by (metis Int_iff \{r\} \rightarrow f \cup m [r + n \mapsto \text{the} (f r) \cdot n] = \{r\} \rightarrow f \uparrow [r + n \mapsto \text{the}
\text{dom}-antirestr \text{dom-eq-singleton-conv empty_iff f’-restr the-dagger-dom-left})
moreover have \(\{x \in \text{dom} [r + n \mapsto \text{the} (f r) \cdot n]. x + \text{the} (\{\{r\} \rightarrow f \uparrow [r + n \mapsto \text{the}
\text{the (f r) \cdot n] x = r\)
= \(\{x \in \text{dom} [r + n \mapsto \text{the} (f r) \cdot n]. x + \text{the} (\{r + n \mapsto \text{the} (f r) \cdot n] x = r\)
by (metis (lifting) l-dagger-apply)
ultimately show \(?\text{thesis by auto}
qed

have \(\cdots = \{\}\)
proof -
have \(\{x \in \text{dom} [r + n \mapsto \text{the} (f r) \cdot n].\)
\(x + \text{the} (\{r + n \mapsto \text{the} (f r) \cdot n] x = r\) = \(\{\}\)
by (smt add_implies_diff comm_monoid_add_class.add.left_neutral
diff_add_zero_dom_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 \(\{x \in \text{dom} (\{r\} \rightarrow f).\)
\end{verbatim}
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

\[ x + \text{the} \left( \left\{ r \right\} - \rho \right) x = r \]
proof -
\begin{align*}
& \text{have} \ \{ x \in \text{dom} \left( \left\{ r \right\} - \rho \right). x + \text{the} \left( \left\{ r \right\} - \rho \right) x = r \}
& \subseteq \{ x \in \text{dom} \left( f \right). x + \text{the} \left( f x \right) = r \}
& \quad \text{by } (\text{smt Collect-cong eq-refl f-in-dom-ar-the-subsame l-dom-ar-not-in-dom r-in-dom}
\text{sep-def sepf})
& \text{also have } \ldots = \{ \}
& \quad \text{by } (\text{smt Collect-empty-eq r-in-dom sep-def sepf})
& \text{finally show } \text{thesis by simp}
& \text{qed}
& \text{ultimately show } \text{thesis by simp}
& \text{qed}
& \text{have } \text{above-shape}: \text{dispose1-above } \left( \left\{ r \right\} - \rho \right) f \cup m \left[ r + n \mapsto \text{the} \left( f r \right) - n \right] \ r \ n = \left[ r + n \mapsto \text{the} \left( f r \right) - n \right]
& \quad \text{unfolding dispose1-above-def}
& \text{proof -}
& \quad \text{have} \ \{ x \in \text{dom} \left( \left\{ r \right\} - \rho \right) \cup m \left[ r + n \mapsto \text{the} \left( f r \right) - n \right], x = r + n \} = \{ r + n \}
& \text{proof -}
& \quad \text{have} \ \{ x \in \text{dom} \left( \left\{ r \right\} - \rho \right) \cup m \left[ r + n \mapsto \text{the} \left( f r \right) - n \right], x = r + n \} = \{ x \in \text{dom} \left( \left\{ r \right\} - \rho \right) \cup m \left[ r + n \mapsto \text{the} \left( f r \right) - n \right], x = r + n \}
& \quad \text{unfolding munion-def by } (\text{subst disjoint-dom-antirestr, simp})
& \text{also have } \ldots = \{ x \in \text{dom} \left( \left\{ r \right\} - \rho \right). x = r + n \} \cup \{ r + n \}
& \quad \text{by auto}
& \text{also have } \ldots = \{ r + n \}
& \text{proof -}
& \quad \text{have} \ \{ x \in \text{dom} \left( \left\{ r \right\} - \rho \right). x = r + n \} = \{ \}
& \quad \text{by } (\text{metis lifting, mono-tags}) \text{Collect-empty-eq gr-n inv l-dom-ar-not-in-dom l-plus-s-not-in-f}
& \text{nat1n r-in-dom)}
& \quad \text{thus } \text{thesis by auto}
& \text{qed}
& \text{finally show } \text{thesis by simp}
& \text{qed}
& \text{moreover have } \{ r + n \} \triangleq \left( \left\{ r \right\} - \rho \right) f \cup m \left[ r + n \mapsto \text{the} \left( f r \right) - n \right] = \left[ r + n \mapsto \text{the} \left( f r \right) - n \right]
& \text{proof } (\text{subst l-dom-r-singleton})
& \text{show } r + n \in \text{dom} \left( \left\{ r \right\} - \rho \right) f \cup m \left[ r + n \mapsto \text{the} \left( f r \right) - n \right]
& \quad \text{by } (\text{smt calculation empty-Collect-eq insert-compr})
& \text{next}
& \text{show } \left[ r + n \mapsto \text{the} \left( \left\{ r \right\} - \rho \right) f \cup m \left[ r + n \mapsto \text{the} \left( f r \right) - n \right] \left( r + n \right) \right]
& \quad = \left[ r + n \mapsto \text{the} \left( f r \right) - n \right]
& \quad \text{by } (\text{metis dagger-upd-dist disjoint-dom-antirestr fun-upd-same munion-def the.simps})
& \text{qed}
& \text{ultimately show } \left\{ x \in \text{dom} \left( \left\{ r \right\} - \rho \right) f \cup m \left[ r + n \mapsto \text{the} \left( f r \right) - n \right], x = r + n \right\} \triangleq \left( \left\{ r \right\} - \rho \right) f \cup m \left[ r + n \mapsto \text{the} \left( f r \right) - n \right]
& \quad \text{by auto}
& \text{qed}
& \text{have } \text{min-loc-shape}: \text{min-loc } \left( \text{dispose1-ext } \left( \left\{ r \right\} - \rho \right) f \cup m \left[ r + n \mapsto \text{the} \left( f r \right) - n \right], r \ n \right) = r
& \text{unfolding dispose1-ext-def}
& \text{proof } (\text{simp add: above-shape below-shape})
& \text{show } \text{min-loc } \left( \left[ r + n \mapsto \text{the} \left( f r \right) - n \right] \cup m \text{Map.empty} \cup m \left[ r \mapsto n \right] \right) = r
& \text{proof -}
have without-empty: \([r + n \mapsto \text{the } (f r) - n] \cup m \Map.empty \cup m \ [r \mapsto n]\)
\[= \ [r + n \mapsto \text{the } (f r) - n] \cup m \ [r \mapsto n]\]
by (metis l-munion-empty-rhs)
then have min-loc \(((r + n \mapsto \text{the } (f r) - n) \cup m [r \mapsto n])
\[= \ min-loc \ ((r + n \mapsto \text{the } (f r) - n) \setunion [r \mapsto n])\]
proof -
have dom \(((r + n \mapsto \text{the } (f r) - n)) \setintersection dom([r \mapsto n]) = \{r+n\} \setintersection \{r\}
by auto
also have ... = {} using nat1n by auto
finally show ?thesis by (simp add: munion-def)
qed
also have ... = \text{min} \ (\text{min-loc } (r + n \mapsto \text{the } (f r) - n)) \ (\text{min-loc } [r \mapsto n])
by (rule min-loc-dagger,simp-all)
also have ... = \text{min} \ (r+n) \ (r) by (simp add: min-loc-singleton)
also have ... = \text{r} by simp
finally show ?thesis using without-empty by simp
qed
qed
have sum-size-shape: sum-size (dispose1-ext \((\{r\} -\triangle f \cup m [r + n \mapsto \text{the } (f r) - n]) \ r \ n)) = \text{the }(f
r)
unfolding dispose1-ext-def
proof (simp add: above-shape below-shape)
show sum-size \(((r + n \mapsto \text{the } (f r) - n) \cup m \emptyset \cup m [r \mapsto n]) = \text{the } (f r)\)
proof -
have without-empty: \(((r + n \mapsto \text{the } (f r) - n) \cup m \Map.empty \cup m [r \mapsto n])
\[= \ [r + n \mapsto \text{the } (f r) - n] \cup m \ [r \mapsto n]\]
by (metis l-munion-empty-rhs)
then have sum-size \(((r + n \mapsto \text{the } (f r) - n) \cup m [r \mapsto n])
\[= \ \text{sum-size } (r + n \mapsto \text{the } (f r) - n)) + \text{sum-size } ([r \mapsto n])\]
apply (subst sum-size-munion, simp-all)
by (metis nat1-def nat1n)
also have ... = \text{the } (f r) - n + n by (simp add: sum-size-singleton)
also have ... = \text{the } (f r) by (metis gr-n le-add-diff-inverse
\text{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 \(f'' = (\text{dom } (\text{dispose1-below } f' r) \cup \text{dom } (\text{dispose1-above } f' r \ n)) -\triangle f'
\cup m \ [\text{min-loc } (\text{dispose1-ext } f' r \ n) \mapsto \text{sum-size } (\text{dispose1-ext } f' r \ n)]\)
then have \(f'' =
\{r+n\} -\triangle f' \cup m \ [\text{min-loc } (\text{dispose1-ext } f' r \ n) \mapsto \text{sum-size } (\text{dispose1-ext } f' r \ n)]\)
by (simp add: f'-restr below-shape above-shape)
also have ... = \((\{r\} -\triangle f) \cup m \ [\text{min-loc } (\text{dispose1-ext } f' r \ n) \mapsto \text{sum-size } (\text{dispose1-ext } f' r \ n)]\)
proof -
have \(\{r+n\} -\triangle f' = \{r+n\} -\triangle (\{r\} -\triangle f \cup m [r + n \mapsto \text{the } (f r) - n])\)
by (simp add: f'-restr)
also have ... = \((r+n) -\triangle ((r) -\triangle (f \cup m [r + n \mapsto \text{the } (f r) - n]))\)
proof(subst l-munion-dom-ar-assoc)
show \(\{r\} \subseteq \text{dom } f\) by (simp add: r-in-dom)
next
show \(\text{dom } f \cap \text{dom } [r + n \mapsto \text{the } (f r) - n] = \{\}\) by (rule disjoint-dom)
next
show \(\{r + n\} -\triangle \{r\} -\triangle (f \cup m [r + n \mapsto \text{the } (f r) - n]) =
\{r + n\} -\triangle \{r\} -\triangle (f \cup m [r + n \mapsto \text{the } (f r) - n])\).
..
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

qed
also have ... = \{r\} -\{ (r+n) -\{ (f \cup m \mid r + n \mapsto the (f r) - n)\}\}
by (metis l-dom-ar-singletons-comm)
also have ... = \{r\} -\{ (r+n) -\{ (f \cup m \mid r + n \mapsto 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 \notin dom f using disjoint-dom by auto
next
show \{r\} = \{r\} -\{ f \} ..
qed
finally show \{\} by simp
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: \(\forall (x::nat). \ x>0 \implies P x \implies P(x+1)\)
  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: \(\forall F F'. \ \text{finite} \ F' \implies \text{finite} \ F' \implies F' \neq \{\} \implies \text{contiguous} \ F' \implies\)

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
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

fix l
assume lindom: l ∈ dom f
show l + the (f l) /∈ dom f
proof -
  have l + the (f l) - 1 ∈ locs-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 \exists 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)) (the (f l)))
    (locs-of (l + the (f l)) (the (f l )))
    by (metis lindom non-abutting)
  then have non-abut-rhs: (\forall 1l:locs-of l (the (f l)). \forall 1l2:locs-of (l + the (f l)) (the (f l + the (f l))))
    1l + 1 < 1l2 ∨ 1l2 + 1 < 1l unfolding non-abut-def by simp
  obtain 1l where 1llocs: 1l ∈ locs-of l (the (f l))
    and 1lshape: 1l = l + (the (f l)) - 1
    by (metis \l + the (f l) - 1 ∈ locs-of l (the (f l)))
  obtain 1l2 where 1l2locs: 1l2 ∈ locs-of (l + the (f l)) (the (f l + the (f l)))
    and 1l2shape: 1l2 = 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 1llocs apply (erule_tac x=1l in ballE)
    using 1l2locs apply (erule_tac x=1l2 in ballE)
    by (simp-all add: 1lshape 1l2shape)
  then show False by auto
  qed
  qed
  qed
  qed
lemma non-abut-Disjoint:
  assumes non-abutting: (\forall l ∈ dom f. \forall l' ∈ dom f. l ≠ l' → non-abut (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 natIf: 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
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

lemma card-contig: card {i::nat. m ≤ i ∧ i < m + l} = l
proof (induct l)
  show base: card {i::nat. m ≤ i ∧ i < m + (0::nat)} = (0::nat)
    by simp
next
  fix l
  assume ind-hyp: card {i::nat. m ≤ i ∧ i < m + l} = l
  show card {i::nat. m ≤ i ∧ i < m + Suc l} = Suc l
    proof -
      have {i::nat. m ≤ i ∧ i < m + Suc l} = \{i::nat. m ≤ i ∧ i < m + l\} ∪ \{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 ⇒ retr0 x = {} ⇒ 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 ∈ dom f ⇒ the (f x) > 0 ⇒
  nat1-map f ⇒ x ∈ (retr0 f)
unfolding retr0-def locs-def locs-of-def by auto

lemma non-empty-nat1-card: finite F ⇒ F ≠ {} ⇒ card F > 0
by auto

lemma eq-locs:
assumes finF: finite F'
  and nonempF: F' ≠ {}
  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 $\exists ! f1. (f0 = \text{retr0 } f1 \land F1-inv f1)$

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 $\exists ! f1$.
proof (induct rule: contig-nonabut-finite-set-induct)
case empty

show $\exists ! f1$. $\{\} = \text{retr0 } f1 \land F1-inv f1$
proof (rule-tac a = empty in ex1I, rule conjI)
  show $\{\} = \text{retr0 } \text{Map.empty}$ by (simp only: retr0-empty)
next
  show F1-inv Map.empty by (simp only: F1-inv-empty)
next
  fix $x$
  assume $\{\} = \text{retr0 } x \land F1-inv x$
  then show $x = \text{empty}$
    by (metis empty-retr0 invF1-nat1-map-weaken)
qed

next
  fix $F$ $F'$
  assume $F'$-finite: finite $F'$
  and notemp: $F' \neq \{\}$
  and $F'$-contig: contiguous $F'$
  and $F'$-nonabut: non-abut $F$ $F'$
  and exist-hyp: $\exists ! f1. F = \text{retr0 } f1 \land F1-inv f1$
from exist-hyp obtain fhook
where ind-hyp-retr: $F = \text{retr0 } fhook$
  and ind-hyp-inv: $F1-inv fhook$
  and ind-hyp-nat1: nat1-map fhook by auto
show $\exists ! f1. F \cup F' = \text{retr0 } f1 \land F1-inv f1$
proof (rule-tac a = (fhook $\cup m (\text{Min } F' \mapsto \text{card } F'))$ in ex1I, rule conjI)

  have nonzerorange: $\forall l \in \text{dom fhook}. (\text{the } (fhook l)) > 0$
    by (metis nat1-def nat1-map-def ind-hyp-nat1)
  have non-intersect: $F \cap F' = \{\}$
    by (metis $F'$-nonabut non-abut-def disjoint-def)
  have domsubsetretr: dom fhook $\subseteq \text{retr0 } fhook$
proof
    fix $x$
    assume indom: $x \in \text{dom fhook}$
    then show $x \in \text{retr0 } fhook$
      proof (rule mapdom-in-retr)
        show $0 < (\text{the } (fhook x))$
          using nonzerorange indom by auto
      next
        show nat1-map fhook by (rule ind-hyp-nat1)
    qed
  qed

then have subsetF: dom fhook $\subseteq F$ using ind-hyp-retr by auto
have min-notin-fhook: Min $F' \notin \text{dom fhook}$
proof -
  have Min $F' \in F'$
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

using notemp F'-finite by auto
then have *: Min F' \notin F
using non-intersect by auto
have Min F' \notin 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 \cup F' = retr0 (f1hook \cup m [Min F' \mapsto card F'])

proof -
have F \cup F' = retr0 (f1hook \cup m [Min(F') \mapsto card F'])
proof -
have dom-extend: dom (f1hook \cup m [Min(F') \mapsto card F'])
  = insert (Min(F')) (dom f1hook)
  by (simp add: l-union-dom min-notin-f1hook)
have nat1-upd-state: nat1-map (f1hook \cup m [Min(F') \mapsto 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 \cup m [Min(F') \mapsto card F'])
  = (\{s::nat\in dom (f1hook \cup m [Min F' \mapsto card F']).
    locs-of s (the ((f1hook \cup m [Min F' \mapsto card F']) s))\})
unfolding retr0-def locs-def by simp
also have (\{s::nat\in dom (f1hook \cup m [Min F' \mapsto card F']).
  locs-of s (the ((f1hook \cup m [Min F' \mapsto card F']) s))\})
  \cup (\{s::nat\in dom f1hook. locs-of s (the ((f1hook \cup m [Min F' \mapsto card F']) s))\})
by (simp only: UN-insert)
also have ... = locs-of (Min F') (the ((f1hook \cup m [Min F' \mapsto card F']) (Min F')))
  \cup (\{s::nat\in dom f1hook. locs-of s (the ((f1hook \cup m [Min F' \mapsto card F']) s))\})
by (simp only: UN-insert)
also have ... = F' \cup (\{s::nat\in dom f1hook. locs-of s (the ((f1hook \cup m [Min F' \mapsto 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 \cup m [Min F' \mapsto card F']) (Min F')) = card F'
apply (subst l-union-apply)
apply (simp add: min-notin-f1hook)
by simp
ultimately show ?thesis by auto
qed
also have ... = (\{s::nat\in dom f1hook. locs-of s (the ((f1hook \cup m [Min F' \mapsto card F']) s))\}) \cup F'
  by blast
also have ... = F \cup F'
proof -
have (\{s::nat\in dom f1hook. locs-of s (the ((f1hook \cup m [Min F' \mapsto card F']) s))\}) = F
proof -
have \forall s \in dom f1hook.
  locs-of s (the (f1hook s)) =
  locs-of s (the ((f1hook \cup m [Min F' \mapsto card F']) s))
apply (subst l-union-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)

  by auto
qed
thus ?thesis by simp
qed

finally show ?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-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
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

qed
show $\min F' + \text{card } F' \notin \text{dom } f1\text{hook}$
proof
assume *: $\min F' + \text{card } F' \in \text{dom } f1\text{hook}$
have $\min F' + \text{card } F' - 1 \in F'$
  by (metis $F'$-contig all-not-in-contiguous-def eq-locs locs-of-finite nat1-card-$F'$
top-locs-of)
obtain l1 where l1shape: $l1 = \min F' + \text{card } F'$ by simp
have l1inF: $l1 \in F$ by (metis * ($l1 = \min F' + \text{card } F'$ set-mp subsetF)
obtain l2 where l2shape: $l2 = \min F' + \text{card } F' - 1$ by simp
have l2inF': $l2 \in F'$ by (metis ($\min F' + \text{card } F' - 1 \in F'$ ($l2 = \min F' + \text{card } F' - 1$))
from $F'$-nonabut have contra: $l1 + 1 < l2 \lor l2 + 1 < l1$
unfolding non-abut-def using l1inF l2inF' by simp
from contra l1shape l2shape show False by auto
qed

next
show $\text{nat1 (card } F')$ by (rule nat1-card-$F'$)
qed
from min-notin-f1hook ind-Disjoint ind-nat1-map
show Disjoint ($f1\text{hook} \cup m [\min F' \mapsto \text{card } F']$)
proof (rule unionm-singleton-Disjoint)
show $\text{nat1 (card } F')$ by (rule nat1-card-$F'$)
next
show disjoint (locs-of ($\min F'$) (card $F'$)) (locs f1\text{hook})
by (metis $F'$-contig $F'$-finite disjoint-def eq-locs ind-hyp-retr
inf-commute non-intersect notemp retr0-def)
qed
qed
qed

fix $x$ assume *: $F \cup F' = \text{retr0} x \land F1\text{-inv } x$
show $x = f1\text{hook} \cup m [\min F' \mapsto \text{card } F']$
proof (rule locs-unique)
show locs $x = \text{locs } (f1\text{hook} \cup m [\min F' \mapsto \text{card } F'])$
  by (metis * ($F \cup F' = \text{retr0} (f1\text{hook} \cup m [\min F' \mapsto \text{card } F'])) retr0-def)
next
show $F1\text{-inv } x$ using * by simp
next
show $F1\text{-inv } (f1\text{hook} \cup m [\min F' \mapsto \text{card } F'])$ by (rule upd-inv)
next
show $x \neq \text{Map.empty}$
by (metis (full-types) * empty-subsetI notemp retr0-empty
sup.right-idem sup-absorb1 sup-commute)
next
show f1\text{hook} \cup m [\min F' \mapsto \text{card } F'] \neq \text{Map.empty}$
by (metis l-munion-singleton-not-empty min-notin-f1hook)
qed
qed
qed
qed
qed

201
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)))
    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 l))
            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

202
LEMMA STRANGESETS: \( B \subseteq A \implies D \subseteq B \implies (A - B) \cup (B - D) = A - D \) 

by auto
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
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} -a f1 by (metis invF1-nat1-map-weaken dom-ar-nat1-map 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
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

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 natf1: nat1-map f1 and findef1: finite (dom f1)
from invf1' show ?thesis
proof
assume sepf1': sep f1' and disjf1': Disjoint f1'
and natf1': nat1-map f1' and findef1': 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-post-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: abovenotempty belowempty l-munion-empty-rhs l-dom-ar-empty-lhs min-loc-singleton sum-size-singleton)
  show \( \text{locs \ (f1} \cup m \ [d1 \mapsto s1]) = \text{locs \ f1} \cup \text{locs-of \ d1 \ s1} \)
proof (rule locs-unionm-singleton)
  show \( \text{nat1 \ s1} \) by (metis nat1s1)
  show \( \text{nat1-map \ f1} \) by (metis nat1f1)
next
  show \( d1 \notin \text{dom \ f1} \) by (rule d1notf1)
qed
qed

next
assumption abovenotempty: dispose1-above f1 d1 s1 \( \not\) \ Map.empty

have abovebelowshape: \( (\text{dom \ (dispose1-below f1 d1)} \cup \text{dom \ (dispose1-above f1 d1 \ s1)}) = \)
\( \{d1+s1\} \)
by (simp add: belowempty l-munion-empty-rhs l-dom-ar-empty-lhs)

proof (rule d1notf1)
  show \( \text{d1-not-dispose-above \ d1-not-dispose-below \ above-dom \ abovenotempty} \)
proof
  have \( \text{min-loc-shape: \ mini-lc (dispose1-ext \ f1 \ d1 \ s1)} = \ d1 \)
  by (metis belowempty l-munion-empty-rhs l-dom-ar-empty-lhs)
  have \( \text{sum-size: \ (dispose1-ext \ f1 \ d1 \ s1) = \ sum-size \ (dispose1-above \ f1 \ d1 \ s1) + s1} \)
  by (simp add: dispose1-ext_def belowempty l-munion-empty-rhs)
next
  have \( \text{sum-size-shape: \ (\text{dom \ (dispose1-below \ f1 \ d1 \ s1)} \cup \text{dom \ (dispose1-above \ f1 \ d1 \ s1)}) =} \)
\( \{d1+s1\} \)
by (simp add: belowempty l-munion-empty-rhs l-dom-ar-empty-lhs)
next

proof (simp add: sum-size-shape min-loc-shape abovebelowshape)
  show \( \text{locs \ ((d1 + s1) -\ a \ f1} \cup m \ [d1 \mapsto \ (the \ (f1 \ (d1 + s1))) + s1]} = \text{locs \ f1} \cup \text{locs-of \ d1 \ s1} \)
proof
  show \( \text{nat1 \ (the \ (f1 \ (d1 + s1)) + s1)} \)
  by (metis nat1-dispose1-ext sum-size-shape)
next
  show \( \text{nat1-map \ ((d1 + s1) -\ a \ f1)} \)
  by (metis dom-ar-nat1-map nat1f1)
next
  show \( d1 \notin \text{dom \ ((d1 + s1) -\ a \ f1)} \)
  by (metis d1notinf1 l-dom-ar-notin-dom-or)
next
  show \( \text{locs \ ((d1 + s1) -\ a \ f1} \cup \text{locs-of \ d1 \ (the \ (f1 \ (d1 + s1))) + s1]} = \text{locs \ f1} \cup \text{locs-of \ d1 \ s1} \)
proof
  have \( \text{locs-of \ (d1 + s1)} \)
  by (metis above-d1s1-in-f1 abovenotempty)

proof
  have \( \text{locs-of \ (d1 + s1)} \) \( \subseteq \text{locs \ f1} \)
  by (metis above-d1s1-in-f1 abovenotempty k-in-locs-iff nat1f1 subsetI)
  moreover have \( \text{locs-of \ d1 \ (the \ (f1 \ (d1 + s1)) + s1)} = \)
\( \text{locs-of \ (d1 + s1)} \)
by (metis above-d1s1-in-f1 abovenotempty)
next
  have \( \text{moreover \ have \ locs-of \ d1 \ (the \ (f1 \ (d1 + s1)) + s1)} = \)
\( \text{locs-of \ d1 \ s1} \)
by (metis above-d1s1-in-f1 abovenotempty)
next
  by (metis nat-add-commute)
next
  also have \( \text{... = \ locs-of \ d1 \ s1} \)
by (metis above-d1s1-in-f1 abovenotempty)
next
  by (metis above-d1s1-in-f1 abovenotempty locs-of-sum-range nat1-map-def nat1f1)

next

natural_text
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

finally show \(?\text{thesis}\) by auto
q\text{ed}
ultimately show \(?\text{thesis}\) by (simp add: g2-subset)
q\text{ed}
q\text{ed}
q\text{ed}
q\text{ed}

next
assume belownotempty: dispose1-below \(f1\ d1\) \(\neq\) Map.empty
from belownotempty have \(\exists\ x. \ x\in\text{dom} f1 \land x + \text{the} (f1 x) = d1\)
proof -
  have dispose1-below \(f1\ d1\) \(\neq\) empty by (rule belownotempty)
  then have \(\{ x \in \text{dom} f1 . x + \text{the}(f1 x) = d1 \} \neq \{\}\)
    by (metis (full-types) dispose1-below-def l-dom-r-nothing)
  thus \(?\text{thesis}\) by (smt empty-Collect-eq)
q\text{ed}
then obtain below where belownf1: below\(\in\)\text{dom} \(f1\)
  and belowplusz1below: below + \text{the} (f1 below) = d1
by metis
then have below-in-dom: below \(\in\) \text{dom}(dispose1-below \(f1\ d1\))
  unfolding dispose1-below-def
proof (subst l-dom-r-iff)
  show below \(\in\) \(\{ x \in \text{dom} f1 . x + \text{the}(f1 x) = d1 \} \cap \text{dom} f1\)
    by (smt Int-Collect belownf1 belowplusz1below inf-commute)
q\text{ed}
have below-shape: dispose1-below \(f1\ d1\) = \([\text{below} \mapsto \text{the} (f1 \text{below})]\)
proof
  fix \(x\)
  show dispose1-below \(f1\ d1\ x\) = \([\text{below} \mapsto \text{the} (f1 \text{below})]\) \(x\)
  proof (simp, intro allI impI conjI)
    from below-in-dom
    show dispose1-below \(f1\ d1\) below = Some (\text{the} (f1 below))
      unfolding dispose1-below-def
proof (subst f-in-dom-r-app-the-elem)
    show below \(\in\) \text{dom} \(f1\) by (rule belownf1)
next
    show below \(\in\) \(\{ x \in \text{dom} f1 . x + \text{the}(f1 x) = d1 \}\)
      by (smt belownf1 belowplusz1below mem-Collect-eq)
q\text{ed}(rule refl)
next
assume znqueteqbelow: \(x \neq\) below
show dispose1-below \(f1\ d1\) \(x\) = None
proof(rule ccontr)
  assume dispose1-below \(f1\ d1\) \(x\) = None then
  have con: \(x\in\) \text{dom} (dispose1-below \(f1\ d1\))
    by auto
  from con have xinmset: \(x\in\) \(\{ x \in \text{dom} f1 . x + \text{the}(f1 x) = d1 \}\)
    unfolding dispose1-below-def
    by (metis (full-types) l-in-dom-dom-r)
then have xinf: \(x\in\) \text{dom} \(f1\) by (simp add: xinmset)
have xeqd1: \(x + \text{the} (f1 x) = d1\)
  by (metis (lifting, mono-tags) mem-Collect-eq xinmset)
from disj1 have *: \text{loc-\text{of} (the (f1 x))} \cap \text{\text{\text{loc-\text{of} below (the (f1 below))}}} = \{\}\n  by (metis znqueteqbelow belownf1 Disjoint-def)
have \text{nat1below}: \text{nat1} \text{ (the } (f1 \text{ below}) \text{) by (metis nat1-map-def nat1f1 belowinf1)}

have \text{nat1x}: \text{nat1} \text{ (the } (f1 x) \text{) by (metis nat1-map-def nat1f1 xinf)}

from \text{xinf xeqd1 belowplusf1below belowinf1 nat1x nat1below}

have **: \text{locs-of } x \text{ (the } (f1 x) \text{) } \cap \text{ locs-of below (the } (f1 \text{ below}) \text{) } \neq \{} \text{ by (metis IntI ex-in-cov top-locs-of)}

from * ** show False by simp
defined

qed

then have \text{dom-below}: \text{dom } (\text{dispose1-below } f1 d1) = \{\text{below}\} \text{ by simp}

have \text{sum-size-below}: \text{sum-size } (\text{dispose1-below } f1 d1) = \text{the } (f1 \text{ below}) \text{ by (simp add: sum-size-singleton below-shape)}

show ?thesis

proof (cases dispose1-above \text{f1 d1 s1} = \text{empty})

assume aboveempty: \text{dispose1-above } \text{f1 d1 s1} = \text{Map.empty}

have \text{abovebelow-shape}: (\text{dom } (\text{dispose1-below } \text{f1 d1})) \cup (\text{dom } (\text{dispose1-above } \text{f1 d1 s1}))

= \{\text{below}\} \text{ by (simp add: aboveempty dom-below)}

have \text{min-loc-shape}: \text{min-loc } (\text{dispose1-ext f1 d1 s1}) = \text{below}

by (metis dom-below insert-not-empty min-below-notempty singleton-iff)

have \text{sum-size-shape}: \text{sum-size } (\text{dispose1-ext f1 d1 s1}) = \text{the } (f1 \text{ below}) + s1

unfolding \text{dispose1-ext-def}

by (simp add: aboveempty l-munion-empty-lhs 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 \text{locs } ((\text{below} - \triangledown f1) \cup m \text{below } \rightarrow \text{the } (f1 \text{ below}) + s1) = \text{locs } f1 \cup \text{locs-of } d1 \text{ s1}

proof (subst \text{locs-unionm-singleton})

show \text{nat1} \text{ (the } (f1 \text{ below}) + s1) \text{ by (metis nat1-dispose1-ext sum-size-shape)}

next

show \text{nat1-map } ((\text{below} - \triangledown f1) \text{ by (metis dom-ar-nat1-map nat1f1)}

next

show \text{below } \notin \text{ dom } ((\text{below} - \triangledown f1) \text{ by (metis f-in-dom-ar-notelem)}

next

show \text{locs } ((\text{below} - \triangledown f1) \cup \text{locs-of below } (\text{the } (f1 \text{ below}) + s1) = \text{locs } f1 \cup \text{locs-of } d1 \text{ s1}

proof -

have \text{locs-of below } (\text{the } (f1 \text{ below})) \subseteq \text{locs } f1

by (metis belowinf1 l-locs-of-within-locs nat1f1)

moreover have \text{*: below } + \text{the } (f1 \text{ below}) = d1 \text{ by (metis belowplusf1below)}

moreover have \text{locs-of below } (\text{the } (f1 \text{ below}) + s1) = \text{locs-of below } (\text{the } (f1 \text{ below}) \cup 

\text{locs-of } d1 \text{ s1}

proof -

have \text{locs-of below } ((\text{the } (f1 \text{ below})) + s1) = \text{locs-of below } (\text{the } (f1 \text{ below})) \cup \text{ (locs-of below } + \text{the } (f1 \text{ below}) \text)) \text{ s1}

by (metis locs-of-sum-range belowinf1 nat1-map-def nat1f1 nat1s1)

also have \text{... } = \text{(locs-of below } (\text{the } (f1 \text{ below})) \cup \text{locs-of } d1 \text{ s1}

by (simp add: *)

finally show ?thesis .

qed

ultimately show ?thesis by (simp add: g2-subset)
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

\[\text{sum-size-shape} \]
\[
\text{dom} \ s_1 \rightarrow \bigcup (\text{proof next assume abovenotempty: dispose1-above } f_1 \ d_1 \ s_1 \neq \text{Map.empty}}
\]
\[
\text{have above-below-shape: (dom dispose1-below } f_1 \ d_1) \cup \text{dom dispose1-above } f_1 \ d_1 s_1) = \{\text{below} \cdot d_1 + s_1}\]
\[
\text{by (metis Un-insert-left above-dom abovenotempty dom-below sup-bot-left)}
\]
\[
\text{have min-loc-shape: min-loc dispose1-ext } f_1 \ d_1 s_1 = \text{below}
\]
\[
\text{by (metis dom-below insert-not-empty min-below-notempty singleton-iff)}
\]
\[
\text{have sum-size-shape: sum-size dispose1-ext } f_1 \ d_1 s_1
\]
\[
= \text{the } (f_1 (d_1 + s_1)) + \text{the } (f_1 \text{ below}) + s_1
\]
\[
\text{proof - have sum-size-above-below: sum-size dispose1-above } f_1 \ d_1 s_1 \cup m \text{ dispose1-below } f_1 \ d_1
\]
\[
= \text{the } (f_1 (d_1 + s_1)) + \text{the } (f_1 \text{ below})
\]
\[
\text{by (simp add: sum-size-munion finite-dispose1-above finite-dispose1-below abovenotempty belownotempty above-sumsize sum-size-below)}
\]
\[
\text{then show ?thesis unfolding dispose1-ext-def}
\]
\[
\text{proof (subst sum-size-munion)}
\]
\[
\text{show finite (dom dispose1-above } f_1 \ d_1 s_1 \cup m \text{ dispose1-below } f_1 \ d_1) \text{ and}
\]
\[
\text{finite (dom } d_1 \rightarrow s_1)\]
\[
\text{by (simp-all add: finite-dispose1-above finite-dispose1-below k-finite-munion)}
\]
\[
\text{next show dispose1-above } f_1 \ d_1 s_1 \cup m \text{ dispose1-below } f_1 \ d_1 \neq \text{empty}
\]
\[
\text{and } [d_1 \rightarrow s_1] \neq \text{empty}
\]
\[
\text{by (auto simp: munion-notempty-right belownotempty)}
\]
\[
\text{next from d1-not-above-below show dom dispose1-above } f_1 \ d_1 s_1 \cup m \text{ dispose1-below } f_1 \ d_1) \cap
\]
\[
\text{dom } d_1 \rightarrow s_1 = \{\}
\]
\[
\text{by simp next show sum-size dispose1-above } f_1 \ d_1 s_1 \cup m \text{ dispose1-below } f_1 \ d_1 + \text{sum-size } d_1 \rightarrow s_1
\]
\[
= \text{the } (f_1 (d_1 + s_1)) + \text{the } (f_1 \text{ below}) + s_1
\]
\[
\text{by (simp add: sum-size-above-below sum-size-singleton)}
\]
\[
\text{qed qed show ?thesis unfolding retr0-def}
\]
\[
\text{proof (simp add: sum-size-shape min-loc-shape above-below-shape, subst locs-unionm-singleton)}
\]
\[
\text{show nat1 (the } f_1 (d_1 + s_1)) + \text{the } (f_1 \text{ below}) + s_1 \text{ by (metis nat1-dispose1-ext sum-size-shape)}
\]
\[
\text{next show nat1-map } \{\text{below} \cdot d_1 + s_1 \rightarrow f_1\} \text{ by (metis dom-ar-nat1-map nat1f1)}
\]
\[
\text{next show below } \notin \text{ dom } \{\text{below} \cdot d_1 + s_1 \rightarrow f_1\} \text{ by (metis insertII l-dom-ar-notin-dom-or)}
\]
\[
\text{next show locs } \{\text{below} \cdot d_1 + s_1 \rightarrow f_1\} \cup \text{locs-of below } (\text{the } f_1 \text{ below}) + s_1 = \text{locs } f_1 \cup \text{locs-of } d_1 \ d_1 \ s_1
\]
\[
\text{proof - have *: (locs } \{\text{below} \cdot d_1 + s_1 \rightarrow f_1\}\} = \text{locs } \{\text{below} \rightarrow (\{d_1 + s_1 \rightarrow f_1\)}
\]
\[
\text{by (metis Un-empty-left Un-insert-left l-dom-ar-accum)}
\]
\[
\text{show ?thesis}
\]
\[
\text{proof (subst *), subst dom-ar-locs)}
\]
\[
\text{show finite } \text{dom } \{\text{d_1 + s_1 \rightarrow f_1}\} \text{ by (metis finite-Diff finitef1 l-dom-dom-ar)}
\]
\[
\text{next show nat1-map } \{\text{d_1 + s_1 \rightarrow f_1}\} \text{ by (metis dom-ar-nat1-map nat1f1)}
\]

209
next
show Disjoint (\{d1 + s1\} -\ a f1) by (metis disjf1 dom-ar-Disjoint)
next
show below \in dom (\{d1 + s1\} -\ a 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\} -\ a f1) - locs-of below (the (\{d1 + s1\} -\ a f1) below)) \cup
locs-of below (the (f1 (d1 + s1)) + the (f1 below) + s1)
= locs f1 \cup locs-of d1 s1
proof (subst dom-ar-locs, rule finitef1, rule nat1f1, rule disjf1)
show d1 + s1 \in dom f1 by (metis above-d1s1-in-f1 abovenotempty)
next

have \*: the (\{d1 + s1\} -\ a f1) below) = the (f1 below)
by (metis belowplusf1below d1notinf1 dom1ff 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\} -\ a f1) below)) \cup
locs-of below (the (f1 (d1 + s1)) + the (f1 below) + s1)
= locs f1 \cup locs-of d1 s1
proof (subst \*)

have ... = (locs-of below (the (f1 below))) \cup
(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 disj2)
apply (metis nat1-def nat1s1)
by simp
also have ... = (locs-of below (the (f1 below)))
\cup (locs-of d1 (s1 + the (f1 (d1 + s1))))
by (metis belowplusf1below nat-add-commute)
also have ... = (locs-of below (the (f1 below)))
\cup (locs-of d1 s1) \cup (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)))
\cup (locs-of (d1 + s1) (the (f1 (d1 + s1)))) \cup (locs-of d1 s1)
by auto
show locs f1 - locs-of (d1 + s1) (the (f1 (d1 + s1))) - locs-of below (the (f1 below)) \cup
locs-of below (the (f1 (d1 + s1)) + the (f1 below) + s1)
= locs f1 \cup locs-of d1 s1
proof (subst ***)
show locs f1 - locs-of (d1 + s1) (the (f1 (d1 + s1))) - locs-of below (the (f1 below)) \cup
(locs-of below (the (f1 below)) \cup
locs-of (d1 + s1) (the (f1 (d1 + s1))) \cup locs-of d1 s1)
= locs f1 \cup locs-of d1 s1
APPENDIX F. HEAP LEMMAS AND PROOFS (IAIN)

proof (subst g3-lemma)
  show locs-of (d1 + s1) (the (f1 (d1 + s1))) ⊆ locs f1 by (metis above-d1s1-in-f1 aboveempty 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 (locf d1 s1 = locs f1 ∪ locs-of d1 s1) by (rule refl)
qed
Appendix G

Earlier Heap models using ZEves

In this chapter we refer to the two technical reports / documents about the AI4FM first attempts at the heap problem. They are available through the AI4FM 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


BIBLIOGRAPHY


