Formal specification and refinement of a Heap specification (V2)

Leo Freitas, Cliff Jones

October 8, 2013

Contents

1 Introduction **1**

2 Some Dependencies **2**
- 2.1 Arithmetic . . . 2
- 2.2 Sets . . . 4
- 2.3 Ranges . . . 4
- 2.4 Minimum . . . 5
- 2.5 Finiteness . . . 5

3 Abstract spec — set of *Loc* **7**
- 3.1 Version 2 of Heap0 . . . 7
- 3.2 Lemmas . . . 8
- 3.3 VCs . . . 9
- 3.4 Playfull lemmas . . . 9
- 3.5 CWG0 original . . . 11

4 Intermediate design — set of *Piece* **13**
- 4.1 Types . . . 13
- 4.2 State and operations . . . 15
- 4.3 Lemmas . . . 16
- 4.4 Refinement relation between $Heap0$ and $Heap1$ . . . 20
- 4.5 VCs . . . 20
- 4.6 Playfull lemmas . . . 23
- 4.7 Refinement theorems . . . 24

5 Proofs **26**
- 5.1 Arithmetics Proofs . . . 26
- 5.2 Set Proofs . . . 27
  - 5.2.1 Bigcup proofs . . . 27
  - 5.2.2 Range proofs . . . 28
  - 5.2.3 Minimum proofs . . . 29
  - 5.2.4 Finiteness proofs . . . 29
- 5.3 CBJ2 Heap0 Proofs . . . 31
  - 5.3.1 WD Proofs . . . 31
  - 5.3.2 Feasibility Proofs . . . 32
  - 5.3.3 Lemmas proofs . . . 33
  - 5.3.4 Playfull lemmas Proofs . . . 34
  - 5.3.5 CWG0 original proofs . . . 37
- 5.4 Commented out proofs . . . 38
- 5.5 CBJ2 Heap1 Proofs . . . 39
  - 5.5.1 WD Proofs . . . 39
  - 5.5.2 Feasibility Proofs . . . 40
  - 5.5.3 Lemmas proofs . . . 46
- 5.6 Heap0 to Heap1 refinement Proofs . . . 60
- 5.7 Unfinished proofs . . . 62
- 5.8 Commented proofs . . . 65

Abstract

Heaps refeniement abstract

Chapter 1

Introduction

This report is about a Z development of a VDM heap model as described in [6, Chapter 7]. The translation strategy from VDM to Z, the input language of the ZEVES theorem prover [3, ?], is described in [8].

We also present developments within the model due to either errors or questionable design decisions. We used the CZT tools[1] to typeset and capture AI4FM proof process data.

[1] See `http://czt.sourceforge.net`

Chapter 2

Some Dependencies

section *arithmetic* **parents** *standard_toolkit*

2.1 Arithmetic

These theorems are simple arithmetic transformations that are often useful when reasoning about non-linear equations in formulae. They are all trivial consequences of integer (Peano's) arithmetic.

Initially, we had the theorems laid out with quantified $\mathbb{Z}$ variables. This is not very helpful as it leads to type-checking proof obligations over the pattern matched expressions to i and j. So, instead of

$$\forall\, i, j : \mathbb{Z} \mid i \geq j \bullet \neg\, i < j$$

we prefer to say simply

$$i \geq j \Rightarrow \neg\, i < j$$

in order to take advantage of the "joker" (place-holder) implicitly (universally) quantified variables, which can only be typed as $\mathbb{A}$ (or $\mathbb{Z}$ in Z/Eves). That is, the former version would lead to a proof obligation that whatever i and j pattern matches to, we would need to show that $i, j \in \mathbb{Z}$, which can be as complex as the expressions for i and j. Using the "joker"-pattern-matching, we avoid this need altogether. It is possible to do it for arithmetic, since it is embedded within the prover. For other situations, one could only hope for weaker type-checking conditions, rather than avoid it altogether, like we are doing here.

[Negate $<$ to $\geq$]

theorem disabled rule lLessNeg
$i \geq j \Rightarrow \neg\, i < j$

[Negate $>$ to $\leq$]

theorem disabled rule lGreaterNeg
$i \leq j \Rightarrow \neg\, i > j$

[Negate $\leq$ to $>$]

theorem disabled rule lLeqNeg
$i > j \Rightarrow \neg\, i \leq j$

[Negate $\geq$ to $<$]

theorem disabled rule lGeqNeg
$i < j \Rightarrow \neg\, i \geq j$

[Flip $<$ to $>$]

theorem rule lLessFlip
$j > i \Rightarrow i < j$

[Flip $>$ to $<$]

theorem disabled rule lGreaterFlip
$j < i \Rightarrow i > j$

[Flip $\leq$ to $\geq$]

theorem disabled rule lLeqFlip
$j \geq i \Rightarrow i \leq j$

[Flip $\geq$ to $\leq$]

theorem disabled rule lGeqFlip
$j \leq i \Rightarrow i \geq j$

For arithmetic "promotion", because we apply it to an operator $(\_ + \_)$-plus that expects $\mathbb{A}$ ($\mathbb{Z}$ in Z/Eves), we do need to add the types for i and j. Otherwise, the decision procedures for arithmetic cannot decide weather to treat i and j as numbers or not.

theorem disabled rule lLessPromote
$\forall i, j : \mathbb{Z} \mid 1 + i \leq j \bullet i < j$

[Promote $>$ into $\geq$]

theorem disabled rule lGreaterPromote
$\forall i, j : \mathbb{Z} \mid i \geq 1 + j \bullet i > j$

$\textbf{section}\ sets\ \textbf{parents}\ standard\_toolkit$

All definitions and lemmas in this section were reused from previous examples within the GC exercises [7, 4, 1, 2, 5, **?**].

2.2 Sets

We also define some new set operators and lemmas. For instance, ZEVES definition of generalised set union is unhelpful because it only matches expressions of the form $x \in \bigcup SS$, whereas if goals contain expressions like $\bigcup SS \subseteq f\ x$, a lot of unnecessary steps are involved. Therefore, we redefine set union to avoid such problems.

$$[XX]$$
$$bigU : \mathbb{P}(\mathbb{P}\ XX) \rightarrow \mathbb{P}\ XX$$
$$\langle\!\langle \text{disabled rule dBigU} \rangle\!\rangle$$
$$\forall SS : \mathbb{P}(\mathbb{P}\ XX) \bullet bigU\ SS = \{v : XX \mid \exists S : \mathbb{P}\ XX \mid S \in SS \bullet v \in S\}$$

Moreover, we prove as an enabled rewrtite rule that both representations are equivalent. Thus, we will not change original models, yet always work with the operator that is easier for proof.

$\textbf{theorem}$ rule dlBigCupAsBigU $[XX]$
$\forall SS : \mathbb{P}(\mathbb{P}\ XX) \bullet \bigcup SS = bigU\ SS$

An easy lemma to have BigU just like $\bigcup$

$\textbf{theorem}$ disabled rule dlInBigU $[XX]$
$\forall SS : \mathbb{P}(\mathbb{P}\ XX) \bullet x \in bigU\ SS \Leftrightarrow (\exists ss : SS \bullet x \in ss)$

$\textbf{theorem}$ disabled rule dlInPowerBigU $[XX]$
$\forall SS : \mathbb{P}(\mathbb{P}\ XX) \mid x \in SS \bullet x \in \mathbb{P}(bigU\ SS)$

$\textbf{theorem}$ rule lCupDiffSubsumption $[X]$
$\forall S : \mathbb{P}\ X;\ x : X \mid x \in S \bullet \{x\} \cup (S \setminus \{x\}) = S$

$\textbf{theorem}$ rule lDiffElimOverNonOverlappingSets $[X]$
$\forall S, T : \mathbb{P}\ X \mid S \cap T = \{\} \bullet S \setminus T = S$

2.3 Ranges

Given the presence of various numeric ranges within the Heap problem, we define some extra (general) lemmas about ranges.

$\textbf{theorem}$ disabled rule dlRangeCapLeft
$\forall A, B, C, D : \mathbb{Z} \mid B < C \bullet (A \mathinner{..} B) \cap (C \mathinner{..} D) = (C \mathinner{..} B)$

$\textbf{theorem}$ disabled rule dlRangeCapRight
$\forall A, B, C, D : \mathbb{Z} \mid D < A \bullet (A \mathinner{..} B) \cap (C \mathinner{..} D) = (A \mathinner{..} D)$

$\textbf{theorem}$ rule dlRangeCapEmpty
$\forall A, B, C, D : \mathbb{Z} \mid B < A \lor D < C \lor B < C \lor D < A \bullet (A \mathinner{..} B) \cap (C \mathinner{..} D) = \{\}$

theorem disabled rule dlRangeSumSubset
$\forall a, b, x, y : \mathbb{N} \mid x \leq a \land a + b \leq x + y \bullet a \mathrel{..} a + b - 1 \subseteq x \mathrel{..} x + y - 1$

theorem disabled rule dlRangeDifference
$\forall A, B, C : \mathbb{Z} \mid A < B \bullet (1 + B \mathrel{..} C) = (A \mathrel{..} C) \setminus (A \mathrel{..} B)$

theorem disabled rule dlRangeSubsumes
$\forall A, B, C, D, x : \mathbb{N} \mid C \leq A \land B \leq D \land x \in A \mathrel{..} B \bullet x \in C \mathrel{..} D$

2.4 Minimum

theorem rule lMinWithinSubset
$\forall S : \mathbb{F}_1\ \mathbb{Z};\ T : \mathbb{P}\ \mathbb{Z} \mid S \in \mathbb{P}\ T \bullet \mathit{min}\ S \in T$

2.5 Finiteness

VDM sets are finite, Z sets are not. Thus, we need to define some extra machinery to handle finite sets within ZEVES smoothly. We also add an inductive principle for finite sets that enables induction proofs over sets, as required by the definitions of *DISPOSE*1 auxiliary functions.

Leave enabled!

theorem rule lCrossFinite2
$A \times B \in \mathbb{F}\,(C \times D) \Leftrightarrow$
$\quad A = \{\,\} \lor B = \{\,\} \lor (A \in \mathbb{F}\ C \land B \in \mathbb{F}\ D)$

Because of size requiring a finite set

theorem rule lFinsetSubset
$\mathit{KnownMember}[\mathbb{F}\ C] \land A \in \mathbb{P}\ \mathit{element} \Rightarrow A \in \mathbb{F}\ C$

theorem disabled lFinsetSubsetKnown
$X \in \mathbb{P}\ Y \land Y \in \mathbb{F}\,Z \Rightarrow X \in \mathbb{F}\,Z$

Leave enabled!

theorem rule lIsFinite
$x \in \mathbb{F}\,X \Rightarrow x \in \mathbb{P}\ X$

theorem disabled lBijectionFinite $[X, Y]$
$\forall A : \mathbb{F}\ X;\ B : \mathbb{P}\ Y \bullet \forall f : A \rightarrowtail B \bullet$
$\quad f \in A \twoheadrightarrow B \land B \in \mathbb{F}\ Y \land \#\ A = \#\ B = \#\ f$

theorem disabled rule lNonMaximalCardEquiv $[X]$
$\forall A : \mathbb{P}\ X \bullet \forall S : \mathbb{F}\ A \bullet \#\ S = (\#\ \_)[A]\ S$

theorem disabled rule lNonMaxDomEquiv $[X, Y]$
$\forall A : \mathbb{P}\ X;\ B : \mathbb{P}\ Y \bullet \forall R : A \leftrightarrow B \bullet \mathrm{dom}[X, Y]\ R = \mathrm{dom}[A, B]\ R$

theorem disabled rule lNonMaxRanEquiv $[X, Y]$
$\forall A : \mathbb{P}\ X;\ B : \mathbb{P}\ Y \bullet \forall R : A \leftrightarrow B \bullet \mathrm{ran}\,[X, Y]\, R = \mathrm{ran}\,[A, B]\, R$

theorem rule lSeqFinite $[X]$
$\forall s : \mathrm{seq}\ X \bullet s \in \mathbb{F}\,(\mathbb{Z} \times X)$

theorem rule lRanSeqFinite $[X]$
$\forall A : \mathbb{P}\ X \bullet \forall s : \mathrm{seq}\ A \bullet \mathrm{ran}\ s \in \mathbb{F}\ A$

generic$(\mathbb{F}_{induc}\ \_)$

$\mathbb{F}_{induc}\, X == \bigcap\{\, A : \mathbb{P}\,\mathbb{P}\, X \mid \varnothing \in A \land (\forall a : A;\ x : X \bullet a \cup \{x\} \in A)\,\}$

Chapter 3

Abstract spec — set of *Loc*

3.1 Version 2 of Heap0

section $Heap2CBJ0$ **parents** $arithmetic, sets$

theorem Loc_ vc_ fsb_ horiz_ def
$\exists Loc : \mathbb{P}\,\mathbb{N} \mid true \bullet true$

Not useful to be disabled

$Loc == \mathbb{N}$

theorem Free0_ vc_ fsb_ horiz_ def
$\exists Free0 : \mathbb{P}\,\mathbb{P}\,Loc \mid true \bullet true$

$Free0 == \mathbb{P}\ Loc$
$Piece \mathrel{\widehat{=}} [\,LOC : Loc;\ SIZE : \mathbb{N}_1\,]$

$locs\_of\ : Piece \rightarrow \mathbb{P}\ Loc$

$\langle\!\langle$ disabled rule dlLocsOfDef $\rangle\!\rangle$
$\forall\, p : Piece \bullet locs\_of\ \ p = p.LOC \mathinner{..} (p.LOC + p.SIZE - 1)$

theorem grule gLocsOfRelType
$locs\_of \in \langle\!| LOC : \mathbb{Z};\ SIZE : \mathbb{Z} |\!\rangle \leftrightarrow \mathbb{P}\ \mathbb{Z}$

theorem rule lLocsOfIsTotal
$\forall\, p : Piece \bullet p \in \operatorname{dom}\ locs\_of$

Heap0

$f0 : Free0$

$true$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad NEW0 \quad}} \\
\Delta Heap0 \\
s? : \mathbb{N}_1 \\
r! : Piece \\
\hline
\exists\, p : Piece \bullet p.SIZE = s? \land locs\_of\,(p) \subseteq f0 \\
s? = r!.SIZE \land f0' = f0 \setminus locs\_of\,(r!) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad DISPOSE0 \quad}} \\
\Delta Heap0 \\
p? : Piece \\
\hline
locs\_of\,(p?) \cap f0 = \varnothing \\
f0' = f0 \cup locs\_of\,(p?) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Init0 \quad}} \\
Heap0\,' \\
memSize? : \mathbb{N}_1 \\
\hline
f0' = 0 \mathinner{\ldotp\ldotp} memSize? - 1 \\
\hline
\end{array}
$$

3.2 Lemmas

theorem grule gLocMaxType
$Loc \in \mathbb{P}\,\mathbb{Z}$

theorem grule gLocType
$Loc \in \mathbb{P}\,\mathbb{N}$

theorem frule fPieceLOCMaxType
$p \in Piece \Rightarrow p.LOC \in \mathbb{Z}$

theorem frule fPieceSIZEMaxType
$p \in Piece \Rightarrow p.SIZE \in \mathbb{Z}$

theorem frule fPieceLOCIsNat
$p \in Piece \Rightarrow p.LOC \geq 0$

theorem frule fPieceSIZEIsNat1
$p \in Piece \Rightarrow p.SIZE > 0$

theorem grule gPieceMaxType
$Piece \in \mathbb{P}\,(\lvert LOC : \mathbb{Z};\ SIZE : \mathbb{Z} \rvert)$

theorem rule lLocsOfResMaxType
$\forall\, p : Piece \bullet locs\_of\,(p) \in \mathbb{P}\,\mathbb{Z}$

theorem disabled rule dlLocsOfProp
$\forall\, p : Piece \bullet locs\_of\,(p) = p.LOC \mathinner{\ldotp\ldotp} p.LOC + p.SIZE - 1$

3.3 VCs

$$\begin{array}{|l} \hline \textit{PieceFSBSig} \\ \quad \textit{Piece} \\ \hline \textit{Piece} \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \textit{Heap0FSBSig} \\ \quad \textit{Heap0} \\ \hline \textit{Heap0} \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \textit{NEW0FSBSig} \\ \quad \textit{Heap0} \\ s? : \mathbb{N}_1 \\ \hline \exists\, q : \textit{Piece} \bullet q.\textit{SIZE} = s? \land \textit{locs\_of}\,(q) \subseteq f0 \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \textit{DISPOSE0FSBSig} \\ \quad \textit{Heap0} \\ p? : \textit{Piece} \\ \hline \textit{locs\_of}\,(p?) \cap f0 = \varnothing \\ \hline \end{array}$$

theorem Piece_ vc_ fsb_ state
$\exists\, \textit{PieceFSBSig} \mid \textit{true} \bullet \textit{true}$

theorem Heap0_ vc_ fsb_ state
$\exists\, \textit{Heap0FSBSig} \mid \textit{true} \bullet \textit{true}$

theorem Init0_ vc_ fsb_ init
$\forall\, \textit{memSize}? : \mathbb{N}_1 \bullet \exists\, \textit{Heap0}' \mid \textit{true} \bullet \textit{Init0}$

theorem NEW0_ vc_ fsb_ pre
$\forall\, \textit{NEW0FSBSig} \mid \textit{true} \bullet \mathbf{pre}\ \textit{NEW0}$

theorem DISPOSE0_ vc_ fsb_ pre
$\forall\, \textit{DISPOSE0FSBSig} \mid \textit{true} \bullet \mathbf{pre}\ \textit{DISPOSE0}$

3.4 Playfull lemmas

$\textit{NEW0DISPOSE0} \mathrel{\widehat{=}} \textit{NEW0} \fatsemi \textit{DISPOSE0}[r!/p?]$

$\textit{Init0NEW0DISPOSE0} \mathrel{\widehat{=}} \textit{Init0} \fatsemi \textit{NEW0} \fatsemi \textit{DISPOSE0}[r!/p?]$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NEW0DISPOSE0FSBSig}} \\
\hline
\mathit{Heap0} \\
s? : \mathbb{N}_1 \\
\hline
\exists\, q : \mathit{Piece} \bullet q.\mathit{SIZE} = s? \land \mathit{locs\_of}\,(q) \subseteq f0 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Init0NEW0DISPOSE0FSBSig}} \\
\hline
\mathit{Init0}[f0/f0'] \\
\mathit{memSize}?, s? : \mathbb{N}_1 \\
\hline
s? \leq \mathit{memSize}? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NEW0DISPOSE0CancelsSig}} \\
\hline
\mathit{Heap0} \\
s? : \mathbb{N}_1 \\
r! : \mathit{Piece} \\
\hline
\exists\, q : \mathit{Piece} \bullet q.\mathit{SIZE} = s? \land \mathit{locs\_of}\,(q) \subseteq f0 \land q = r! \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Init0NEW0DISPOSE0CancelsSig}} \\
\hline
\mathit{Heap0}' \\
\mathit{memSize}?, s? : \mathbb{N}_1 \\
r! : \mathit{Piece} \\
\hline
s? = r!.\mathit{SIZE} \\
r!.\mathit{LOC} + s? \leq \mathit{memSize}? \\
\hline
\end{array}
$$

theorem disabled rule dlArithAux

$\forall\, a, x, b : \mathbb{N};\ m : \mathbb{N}_1 \mid a + b \leq m \land a \leq x \land x < a + b \bullet x < m$

theorem NEW0DISPOSE0_vc_fsb_pre

$\forall\, \mathit{NEW0DISPOSE0FSBSig} \bullet \mathbf{pre}\ \mathit{NEW0DISPOSE0}$

theorem Init0NEW0DISPOSE0_vc_fsb_pre

$\forall\, \mathit{Init0NEW0DISPOSE0FSBSig} \bullet \mathbf{pre}\ \mathit{Init0NEW0DISPOSE0}$

theorem NEW0DISPOSE0Cancels

$\forall\, \mathit{NEW0DISPOSE0CancelsSig} \bullet \mathit{NEW0DISPOSE0} \Leftrightarrow \Xi \mathit{Heap0}$

theorem Init0NEW0DISPOSE0Cancels

$\forall\, \mathit{Init0NEW0DISPOSE0CancelsSig} \bullet \mathit{Init0NEW0DISPOSE0} \Leftrightarrow \mathit{Init0}$

3.5 CWG0 original

$\mathbf{relation}(\mathbf{hasSeq}\,\_)$
$\mathbf{relation}(\mathbf{isSequential}\,\_)$

$\mathbf{isSequential}\,\_ == \{s : \mathrm{seq}\ Loc \mid \exists\, i, j : \mathbb{N} \bullet \mathrm{ran}\ s = i \mathinner{..} j\}$
$\mathbf{hasSeq}\,\_ == \{s : \mathrm{seq}\ Loc;\ size : \mathbb{N};\ q : Free0 \mid \mathbf{isSequential}\ s \land \mathrm{ran}\ s \subseteq q \land \#\ s = size\}$

NEWCWG0

$\Delta Heap0$
$s? : \mathbb{N}$
$r! : \mathbb{F}\ Loc$

$\exists\, s : \mathrm{seq}\ Loc \bullet \mathbf{hasSeq}\,(s, s?, f0) \land f0' = f0 \setminus r! \land r! = \mathrm{ran}\ s$

DISPOSECWG0

$\Delta Heap0$
$r? : \mathbb{F}\ Loc$

$f0 \cap r? = \varnothing$
$f0' = f0 \cup r?$

NEWCWG0FSBSig

$Heap0$
$s? : \mathbb{N}$

$\exists\, t : \mathrm{seq}\ Loc \bullet \mathbf{hasSeq}\,(t, s?, f0)$

DISPOSECWG0FSBSig

$Heap0$
$r? : \mathbb{F}\ Loc$

$f0 \cap r? = \varnothing$

theorem NEWCWG0_ vc_ fsb_ pre
$\forall\, NEWCWG0FSBSig \mid true \bullet \mathbf{pre}\ NEWCWG0$

theorem DISPOSECWG0_ vc_ fsb_ pre
$\forall\, DISPOSECWG0FSBSig \mid true \bullet \mathbf{pre}\ DISPOSECWG0$

$NEWCWG0DISPOSECWG0 \mathrel{\widehat=} NEWCWG0 \mathbin{⨾} DISPOSECWG0[r!/r?]$

$Init0NEWCWG0DISPOSECWG0 \mathrel{\widehat=} Init0 \mathbin{⨾} NEWCWG0 \mathbin{⨾} DISPOSECWG0[r!/r?]$

$$\begin{array}{l}
\textit{NEWCWG0DISPOSECWG0CancelsSig} \\
\hline
\mathit{Heap}0 \\
s? : \mathbb{N}_1 \\
r! : \mathbb{F}\ \mathit{Loc} \\
\hline
\exists t : \mathrm{seq}\ \mathit{Loc} \bullet \mathbf{hasSeq}\,(t, s?, f0) \land r! = \mathrm{ran}\ t \\
\hline
\end{array}$$

theorem NEWCWG0DISPOSECWG0Cancels
$\forall\ \mathit{NEWCWG0DISPOSECWG0CancelsSig} \bullet \mathit{NEWCWG0DISPOSECWG0} \Leftrightarrow \Xi \mathit{Heap}0$

Declarations	**This Chapter**	**Globally**
Unboxed items	12	16
Axiomatic definitions	1	1
Generic axiomatic defs.	0	1
Schemas	17	17
Generic schemas	0	0
Theorems	26	58
Proofs	0	0
Total	**56**	**93**

Table 3.1: Summary of Z declarations for Chapter 3.

Chapter 4

Intermediate design — set of *Piece*

This is the specification of $Heap1$ of CBJ model version 2

$\textbf{section}\ Heap2CBJ1\ \textbf{parents}\ Heap2CBJ0, sets$

$\textbf{relation}(sep\ \_)$
$\textbf{relation}(\_\ \textbf{before}\ \_)$
$\textbf{relation}(\_\ \textbf{abutt}\ \_)$
$\textbf{relation}(\_\ \textbf{fuse}\ \_)$
$\textbf{relation}(\textbf{unique}\ \_)$
$\textbf{relation}(\_\ \textbf{wellplaced}\ \_)$

4.1 Types

$\_\ \textbf{before}\ \_ == \{p1, p2 : Piece \mid p1.LOC + p1.SIZE < p2.LOC\}$
$\textbf{unique}\ \_ == \{fr : \mathbb{P}\ Piece \mid \forall p1, p2 : fr \mid p1.LOC = p2.LOC \bullet p1 = p2\}$
$\_\ \textbf{wellplaced}\ \_ == \{p1, p2 : Piece \mid \textbf{unique}\ \{p1, p2\} \land (p1\ \textbf{before}\ p2 \lor p2\ \textbf{before}\ p1)\}$
$\_\ \textbf{fuse}\ \_ == \{p1, p2 : Piece \mid p1.LOC + p1.SIZE = p2.LOC\}$
$\_\ \textbf{abutt}\ \_ == \{p1, p2 : Piece \mid p1\ \textbf{fuse}\ p2 \lor p2\ \textbf{fuse}\ p1\}$
$sep\ \_ == \{fr : \mathbb{P}\ Piece \mid \forall p1, p2 : fr \mid p1.LOC < p2.LOC \bullet p1\ \textbf{before}\ p2\}$

theorem Free1_ vc_ fsb_ horiz_ def
$\exists Free1 : \mathbb{P}\{ps : \mathbb{P}\ Piece \mid ps \in \mathbb{F}\ Piece \land sep(ps) \land \textbf{unique}(ps)\} \mid true \bullet true$

$Free1 == \{ps : \mathbb{P}\ Piece \mid ps \in \mathbb{F}\ Piece \land sep\ (ps) \land \textbf{unique}\ (ps)\}$

theorem frule fFree1ElemMaxType
$f \in Free1 \Rightarrow f \in \mathbb{P}\ \langle\!|LOC : \mathbb{Z};\ SIZE : \mathbb{Z}|\!\rangle$

theorem frule fFree1ElemType
$f \in Free1 \Rightarrow f \in \mathbb{P}\ Piece$

theorem frule fFree1ElemFinType
$f \in Free1 \Rightarrow f \in \mathbb{F}\ Piece$

$$
\begin{array}{|l}
locs : \mathbb{P}\ Piece \rightarrow \mathbb{P}\ Loc \\
\hline
\langle\!\langle \text{disabled rule dlLocsDef} \rangle\!\rangle \\
\forall f : \mathbb{P}\ Piece \bullet locs\ f = \bigcup \{ p : Piece \mid p \in f \bullet locs\_of\ (p) \}
\end{array}
$$

theorem grule gLocsRelType
$locs \in \mathbb{P}\,(\langle\!| LOC : \mathbb{Z};\ SIZE : \mathbb{Z} |\!\rangle) \leftrightarrow \mathbb{P}\,\mathbb{Z}$

theorem rule lLocsIsTotal
$\forall f : \mathbb{P}\ Piece \bullet f \in \mathrm{dom}\ locs$

$$
\begin{array}{|l}
allLocs : \mathbb{P}\ Piece \rightarrow \mathbb{P}\ Loc \\
\hline
\langle\!\langle \text{disabled rule dlAllLocs} \rangle\!\rangle \\
\forall ps : \mathbb{P}\ Piece \bullet allLocs\,(ps) = \{ p : Piece \mid p \in ps \bullet p.LOC \}
\end{array}
$$

theorem grule gAllLocsRelType
$allLocs \in \mathbb{P}\,(\langle\!| LOC : \mathbb{Z};\ SIZE : \mathbb{Z} |\!\rangle) \leftrightarrow \mathbb{P}\,\mathbb{Z}$

theorem rule lAllLocsIsTotal
$\forall f : \mathbb{P}\ Piece \bullet f \in \mathrm{dom}\ allLocs$

$$
\begin{array}{|l}
minLoc : \mathbb{P}_1\ Piece \rightarrow Loc \\
\hline
\langle\!\langle \text{disabled rule dlMinLoc} \rangle\!\rangle \\
\forall ps : \mathbb{P}_1\ Piece \bullet minLoc\,(ps) = min\,(allLocs\,(ps))
\end{array}
$$

theorem grule gMinLocRelType
$minLoc \in \mathbb{P}\,(\langle\!| LOC : \mathbb{Z};\ SIZE : \mathbb{Z} |\!\rangle) \leftrightarrow \mathbb{Z}$

theorem rule lMinLocIsTotal
$\forall f : \mathbb{P}_1\ Piece \bullet f \in \mathrm{dom}\ minLoc$

theorem rule lMinLocResIsNat
$\forall f : \mathbb{P}_1\ Piece \bullet minLoc\ f \geq 0$

$$
\begin{array}{|l}
sumSize : \mathbb{F}\ Piece \rightarrow \mathbb{N} \\
\hline
\langle\!\langle \text{disabled rule dlSumSizeBase} \rangle\!\rangle \\
sumSize\ \{\} = 0 \\
\langle\!\langle \text{disabled rule dlSumSizeInduct} \rangle\!\rangle \\
\forall p : Piece;\ ps : \mathbb{F}\ Piece \bullet sumSize\,(ps \cup \{p\}) = p.SIZE + sumSize(ps)
\end{array}
$$

theorem grule gSumSizeRelType
$sumSize \in \mathbb{P}\,(\langle\!| LOC : \mathbb{Z};\ SIZE : \mathbb{Z} |\!\rangle) \leftrightarrow \mathbb{Z}$

theorem rule lSumSizeIsTotal
$\forall f : \mathbb{F}\ Piece \bullet f \in \mathrm{dom}\ sumSize$

theorem rule lSumSizeResIsNat
$\forall f : \mathbb{F}\ Piece \bullet sumSize\ f \geq 0$

theorem rule lSumSizeResIsNat1
$\forall f : \mathbb{F}\ Piece \mid \neg\ f = \{\} \bullet sumSize\ f \geq 1$

4.2 State and operations

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Heap1}} \\
\hline
f1 : \textit{Free1} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Init1}} \\
\hline
\textit{Heap1}' \\
\textit{memSize}? : \mathbb{N}_1 \\
\hline
f1' = \{\theta\, \textit{Piece}[\textit{LOC} := 0, \textit{SIZE} := \textit{memSize}?]\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NEW1}} \\
\hline
\Delta \textit{Heap1} \\
s? : \mathbb{N}_1 \\
r! : \textit{Piece} \\
\hline
\exists\, q : \textit{Piece} \bullet q \in f1 \land q.\textit{SIZE} \geq s? \\
\exists\, p : \textit{Piece} \bullet p \in f1 \land \\
\qquad p.\textit{SIZE} \geq s? \land \\
\qquad r! = \theta\, \textit{Piece}[\textit{LOC} := p.\textit{LOC}, \textit{SIZE} := s?] \land \\
\qquad (p.\textit{SIZE} = s? \Rightarrow f1' = (f1 \setminus \{p\})) \land \\
\qquad (p.\textit{SIZE} > s? \Rightarrow f1' = (f1 \setminus \{p\}) \cup \\
\qquad\qquad \{\theta\, \textit{Piece}[\textit{LOC} := p.\textit{LOC} + s?, \textit{SIZE} := p.\textit{SIZE} - s?]\}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NEW1Exact}} \\
\hline
\Delta \textit{Heap1} \\
s? : \mathbb{N}_1 \\
r! : \textit{Piece} \\
\hline
\exists\, t : \textit{Piece} \bullet t \in f1 \land t.\textit{SIZE} = s? \land \\
\qquad r! = \theta\, \textit{Piece}[\textit{LOC} := t.\textit{LOC}, \textit{SIZE} := s?] \land \\
\qquad f1' = (f1 \setminus \{t\}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NEW1Bigger}} \\
\hline
\Delta \textit{Heap1} \\
s? : \mathbb{N}_1 \\
r! : \textit{Piece} \\
\hline
\exists\, t, \textit{rem} : \textit{Piece} \bullet t \in f1 \land t.\textit{SIZE} > s? \land \\
\qquad r! = \theta\, \textit{Piece}[\textit{LOC} := t.\textit{LOC}, \textit{SIZE} := s?] \land \\
\qquad \textit{rem} = \theta\, \textit{Piece}[\textit{LOC} := t.\textit{LOC} + s?, \textit{SIZE} := t.\textit{SIZE} - s?] \land \\
\qquad f1' = (f1 \setminus \{t\}) \cup \{\textit{rem}\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NEW1Leo}} \\
\hline
\Delta \textit{Heap1} \\
s? : \mathbb{N}_1 \\
r! : \textit{Piece} \\
\hline
\textit{NEW1Exact} \lor \textit{NEW1Bigger} \\
\hline
\end{array}
$$

theorem disabled rule dlNEW1Equiv
$NEW1 \Leftrightarrow NEW1Leo$

This is okay, but makes the proof slighly lengthier because of the one-point-rule applied to *join*. Will define an slighly alternative version

$DISPOSE1$

$\Delta Heap1$
$p? : Piece$

$locs\_of\ (p?) \cap locs\ (f1) = \varnothing$
$\exists\, join, abt : \mathbb{P}\ Piece;\ np : Piece \bullet$
$\quad abt = \{q : Piece \mid q \in f1 \land p?\ \mathbf{abutt}\ q\} \land$
$\quad join = \{p?\} \cup abt \land$
$\quad np = \theta\, Piece[LOC := minLoc\,(join), SIZE := sumSize\,(join)] \land \qquad f1' = (f1 \setminus join) \cup \{np\}$

$DISPOSE1Join$

$fr : Free1$
$np2, p2 : Piece$
$join2 : \mathbb{P}\ Piece$
$abt2 : \mathbb{P}\ Piece$

$abt2 = \{r : Piece \mid r \in fr \land p2\ \mathbf{abutt}\ r\}$
$join2 = \{p2\} \cup abt2$
$np2 = \theta\, Piece[LOC := minLoc\,(join2), SIZE := sumSize\,(join2)]$

theorem dlDISPOSE1JoinWitness
$\forall\, Heap1;\ p? : Piece \bullet \exists\, DISPOSE1Join \bullet fr = f1 \land p2 = p?$

$DISPOSE1Leo$

$\Delta Heap1$
$p? : Piece$

$locs\_of\ (p?) \cap locs\ (f1) = \varnothing$
$\exists\, DISPOSE1Join \bullet fr = f1 \land p2 = p? \land f1' = (f1 \setminus join2) \cup \{np2\}$

theorem disabled rule dlDISPOSE1Equiv
$DISPOSE1 \Leftrightarrow DISPOSE1Leo$

4.3 Lemmas

theorem disabled rule dlInFree1
$f \in Free1 \Leftrightarrow (f \in \mathbb{P}\ Piece \land f \in \mathbb{F}\ Piece \land sep\,(f) \land \mathbf{unique}\,(f))$

theorem disabled rule dlInInvFree1Unique
$\mathbf{unique}\,(fr) \Leftrightarrow (fr \in \mathbb{P}\ Piece \land \forall\, p1, p2 : Piece \mid p1 \in fr \land p2 \in fr \land p1.LOC = p2.LOC \bullet p1 = p2)$

theorem disabled rule dlInInvFree1Sep
$sep\,(fr) \Leftrightarrow (fr \in \mathbb{P}\ Piece \land \forall\, p1, p2 : Piece \mid p1 \in fr \land p2 \in fr \land p1.LOC < p2.LOC \bullet p1\ \mathbf{before}\ p2)$

Useful in various cases

theorem rule lFree1Empty
$\{\} \in Free1$

These lemmas are useful to avoid expanding *Piece*

theorem frule fPieceLOCType
$p \in Piece \Rightarrow p.LOC \in Loc$

theorem rule lFree1ReductionInType
$\forall f : Free1;\ g : \mathbb{P}\ Piece \bullet f \setminus g \in Free1$

theorem disabled rule dlPieceBefore
$\forall p1, p2 : Piece \bullet p1 \ \mathbf{before}\ p2 \Leftrightarrow (p1.LOC + p1.SIZE < p2.LOC)$

theorem disabled rule dlPieceWellPlaced
$\forall p1, p2 : Piece \bullet p1 \ \mathbf{wellplaced}\ p2 \Leftrightarrow (\neg\ p1.LOC = p2.LOC \land p1 \ \mathbf{before}\ p2 \lor p2 \ \mathbf{before}\ p1)$

theorem disabled rule dlPieceExcludedMiddle
$\forall p1, p2 : Piece \mid p1 \ \mathbf{before}\ p2 \bullet \neg\ p2 \ \mathbf{before}\ p1$

theorem rule lPieceNotBeforeItself
$\forall p : Piece \bullet \neg\ p \ \mathbf{before}\ p$

theorem disabled rule dlFree1UnitUnionUniqueProp
$$\begin{aligned}&\forall f : Free1;\ p : Piece \bullet\\ &\qquad \mathbf{unique}\,(f \cup \{p\})\\ &\qquad\quad \Leftrightarrow\\ &\qquad (\forall q : Piece \mid q \in f \land q.LOC = p.LOC \bullet q.SIZE = p.SIZE)\end{aligned}$$

theorem rule lFree1UnitUnionUnique
$\forall f : Free1;\ p : Piece \mid \neg\ p.LOC \in allLocs\,(f) \bullet \mathbf{unique}\,(f \cup \{p\})$

theorem rule lFree1UnitUnionInType
$$\begin{aligned}&\forall f : Free1;\ t : Piece \mid \neg\ t.LOC \in allLocs\,(f) \land\\ &\qquad (\forall r : Piece \bullet (r \in f \land t.LOC < r.LOC \Rightarrow t \ \mathbf{before}\ r) \land\\ &\qquad\quad (r \in f \land r.LOC < t.LOC \Rightarrow r \ \mathbf{before}\ t)) \bullet\\ &\qquad\qquad f \cup \{t\} \in Free1\end{aligned}$$

```
\begin{theoremold}{rule lFree1UnitUnionInType2}
\forall f: Free1; t: Piece @ f \cup \{ t \} \in Free1 \iff (\forall q: Piece | q \in f @ q \wellplaced t)
\end{theoremold}
\begin{theoremold}{rule lFree1UnitUnionInType3}
\forall f: Free1; t: Piece | \\
\t1 \lnot t.LOC \in allLocs~(f) \land (\forall q: Piece | q \in f @ q \wellplaced t) @ \\
\t2 f \cup \{ t \} \in Free1
\end{theoremold}
\begin{theoremold}{rule lFree1UnitUnionInType4}
\forall f: Free1; t: Piece | \lnot t.LOC \in allLocs~(f) \land \\
\t1 (\forall r: Piece | r \in f @ t \before r \lor r \before t) @ \\
\t2 f \cup \{ t \} \in Free1
\end{theoremold}
```

theorem rule dlLocsOfCapEmpty
$\forall p, q : Piece \mid p.LOC + p.SIZE \leq q.LOC \lor q.LOC + q.SIZE \leq p.LOC \bullet$
$locs\_of\ (p) \cap locs\_of\ (q) = \{\}$

theorem rule lLocsDistUnitDiff
$\forall f : Free1 \bullet \forall p : f \bullet locs\ (f \setminus \{p\}) = locs\ (f) \setminus locs\_of\ (p)$

theorem rule lLocsDistUnitCup
$\forall f : Free1 \bullet \forall p : f \bullet locs\ (\{p\} \cup f) = locs\ (f) \cup locs\_of\ (p)$

Useful for the correctness proof of NEW1 for the case that f1' is within f0'

theorem disabled rule dlInLocs
$\forall f : Free1 \bullet x \in locs\, f \Leftrightarrow$
$(\exists q : Piece \bullet q \in f \land x \in locs\_of\ \ q)$

Having it as IFF makes for much better/useful lemma as it can be used on hypothesis as well.

theorem disabled rule dlLocsCapEmpty
$\forall f : Free1;\ a : Piece \bullet locs\_of\ (a) \cap locs\ (f) = \{\} \Leftrightarrow$
$(\forall t : Piece \mid t \in f \bullet locs\_of\ \ a \cap locs\_of\ \ t = \{\})$

theorem frule fDISPOSEJoinWithinHeap
$\forall DISPOSE1Join \bullet (join2 \setminus \{p2\}) \in \mathbb{P}\ fr$

theorem frule fDISPOSEJoinAbtWithinHeap
$\forall DISPOSE1Join \bullet abt2 \in \mathbb{P}\ fr$

theorem rule lInAllLocs
$\forall f : \mathbb{P}\ Piece;\ t : Piece \mid t \in f \bullet t.LOC \in allLocs\, f$

theorem rule lAllLocsWithin
$\forall f, g : \mathbb{P}\ Piece \mid f \in \mathbb{P}\ g \bullet allLocs\ (f) \in \mathbb{P}\ allLocs\ (g)$

theorem rule lAllLocResIsNotEmpty
$\forall f : \mathbb{P}_1\ Piece \bullet \neg\ allLocs\, f = \{\}$

For the general case where $f \cap g \neq \varnothing$ and the element is in g but not in f, we can't use the uniqueness invariant. For instance if f contained a Piece $(L0, S3)$ and no other locartion with location 0, and g contained a Piece $(L0, S2)$ and no other Piece with location 0, then the equality doesn't hold. Where the lemma comes from, $DISPOSE1Join$, the precondition ensures that that is the case because $p?$ is not within the locations of f. But in general, we need the side condition.

theorem rule lAllLocsDiff
$\forall f : Free1;\ g : \mathbb{P}\ Piece \mid g \in \mathbb{P}\ f \bullet$
$allLocs\ (f \setminus g) = allLocs\, f \setminus allLocs\, g$

See lPayDetailsBindingsSubsetPayDetailsSpace in Mondex for the way to prove this. It's long winded and tedious, but it works.

theorem rule lAllLocsIsFinset1
$\forall f : \mathbb{F}_1\ Piece \bullet allLocs\, f \in \mathbb{F}_1\ \mathbb{Z}$

¡¡¡¡¡¡ .mine this obviously apply when f = g, hence the corolary $minLoc\, f \in allLocs\, f$ ======= this obviously apply when $f = g$, hence the corolary $minLoc\, f \in allLocs\, f$ ¿¿¿¿¿¿¿ .r1350

theorem rule lMinLocsWithnAllLocs
$\forall f : \mathbb{F}_1\ Piece;\ g : \mathbb{P}\ Piece \mid (f \in \mathbb{P}\ g) \bullet minLoc\,(f) \in allLocs\,(g)$

theorem disabled rule dlInFuse
$(p\ \mathbf{fuse}\ q) \Leftrightarrow (p \in Piece \land q \in Piece \land p.LOC + p.SIZE = q.LOC)$

theorem disabled rule dlFuseExcludedMiddle
$\forall p, q : Piece \mid p\ \mathbf{fuse}\ q \bullet \neg\ q\ \mathbf{fuse}\ p$

theorem disabled rule dlInAbutt
$(p\ \mathbf{abutt}\ q) \Leftrightarrow (p \in Piece \land q \in Piece \land p\ \mathbf{fuse}\ q \lor q\ \mathbf{fuse}\ p)$

theorem rule lNoSelfAbutt
$\forall p : Piece \bullet \neg\ p\ \mathbf{abutt}\ p$

theorem rule lAllLocsUnit
$\forall p : Piece \bullet allLocs\,\{p\} = \{p.LOC\}$

theorem rule lMinLocUnit
$\forall p : Piece \bullet minLoc\,\{p\} = p.LOC$

theorem rule lSumSizeUnit
$\forall p : Piece \bullet sumSize\,\{p\} = p.SIZE$

NOTE-1: These are exploratory lemmas trying to infer what are the properties for each case over join. Later (or rather backwards), through the proof of the auxiliary lemma for **pre** $DISPOSE1$ (lDISPOSE1FSBAuxLemma), a new (simpler?) side condition emerges, hence the new generation of lemmas

theorem dlDISPOSE1JoinNoAbutt
$\forall DISPOSE1Join \mid \neg\,(\exists w : fr \bullet p2\ \mathbf{abutt}\ w) \bullet join2 = \{p2\}$

theorem dlDISPOSE1JoinAbuttBefore
$\forall DISPOSE1Join \mid \neg\,(\exists t : fr \bullet p2\ \mathbf{fuse}\ t) \bullet$
$join2 = \{p2\} \cup \{l : Piece \mid l \in fr \land l\ \mathbf{fuse}\ p2\}$

theorem dlDISPOSE1JoinAbuttAfter
$\forall DISPOSE1Join \mid \neg\,(\exists t : fr \bullet t\ \mathbf{fuse}\ p2) \bullet$
$join2 = \{p2\} \cup \{r : Piece \mid r \in fr \land p2\ \mathbf{fuse}\ r\}$

This although right, doesn't help the proof either because it has the wrong shape again. The goal talks about properties of a known piece (r) (i.e. NOTE-1: goal features?)

theorem OLD-IGNORED dlDISPOSE1JoinNoAbuttV2
$\forall\, DISPOSE1Join \mid abt2 = \{\} \bullet join2 = \{p2\}$

theorem OLD-IGNORED dlDISPOSE1JoinAbuttUnique
$\forall\, DISPOSE1Join \mid \neg\, abt2 = \{\} \bullet \exists_1\, a : Piece \bullet a \in fr \land a \textbf{ fuse } p2a \Rightarrow abt2 = \{$

theorem OLD-IGNORED dlDISPOSE1JoinAbuttUnique
$\forall\, DISPOSE1Join \mid \neg\, abt2 = \{\} \bullet abt2 = \{labt, rabl \mid \exists\, q : Piece \mid\bullet \forall\, labt, rabt : Piece \mid \neg\, abt2 = \{\}$
$labt \textbf{ fuse } p2 \bullet abt2 = \{labt, rabt\}$

4.4 Refinement relation between $Heap0$ and $Heap1$

RetrFree0Free1
$Heap0$
$Heap1$

$f0 = locs\, f1$

4.5 VCs

Heap1FSBSig
$Heap1$

$Heap1$

Init1FSBSig
$memSize? : \mathbb{N}_1$

$memSize? > 0$

This precondition is open (and sufficient), yet the concrete one to choose for *res* and *rem* are quite prescribed (I think): they need at least to start at the same place (i.e. $res.LOC = p.LOC$), otherwise one would get two pieces remaining rather than one, which is not the case in the postcondition. So this is kind of implicit in the precondition.

NEW1FSBSig
$Heap1$
$s? : \mathbb{N}_1$

$\exists\, q : Piece \bullet q \in f1 \land q.SIZE \geq s?$

NEW1ExactFSBSig
$Heap1$
$s? : \mathbb{N}_1$

$\exists\, q : Piece \bullet q \in f1 \land q.SIZE = s?$

$$\begin{array}{l}
\textit{NEW1BiggerFSBSig} \\
\quad \textit{Heap1} \\
s? : \mathbb{N}_1 \\
\hline
\exists\, q : \textit{Piece} \bullet q \in f1 \land q.\textit{SIZE} > s?
\end{array}$$

$$\begin{array}{l}
\textit{NEW1LeoFSBSig} \\
\quad \textit{Heap1} \\
s? : \mathbb{N}_1 \\
\hline
\exists\, q : \textit{Piece} \bullet q \in f1 \land q.\textit{SIZE} \geq s?
\end{array}$$

$$\begin{array}{l}
\textit{DISPOSE1FSBSig} \\
\quad \textit{Heap1} \\
p? : \textit{Piece} \\
\hline
\textit{locs_of}\ (p?) \cap \textit{locs}\ (f1) = \varnothing
\end{array}$$

$$\begin{array}{l}
\textit{DISPOSE1JoinFSBSig} \\
\quad \textit{DISPOSE1Join} \\
\hline
\textit{DISPOSE1Join}
\end{array}$$

$$\begin{array}{l}
\textit{DISPOSE1LeoFSBSig} \\
\quad \textit{Heap1} \\
p? : \textit{Piece} \\
\hline
\textit{locs_of}\ (p?) \cap \textit{locs}\ (f1) = \varnothing
\end{array}$$

$$\begin{array}{l}
\textit{RetrFree0Free1FSBSig} \\
\quad \textit{Heap0} \\
\textit{Heap1} \\
\hline
\textit{RetrFree0Free1}
\end{array}$$

theorem Heap1_ vc_ fsb_ state
$\exists\, \textit{Heap1FSBSig} \mid \textit{true} \bullet \textit{true}$

theorem Init1_ vc_ fsb_ init
$\forall\, \textit{Init1FSBSig} \bullet \exists\, \textit{Init1} \bullet \textit{true}$

theorem DISPOSE1Join_ vc_ fsb_ state
$\exists\, \textit{DISPOSE1JoinFSBSig} \mid \textit{true} \bullet \textit{true}$

theorem RetrFree0Free1_ vc_ fsb_ state
$\exists\, \textit{RetrFree0Free1FSBSig} \mid \textit{true} \bullet \textit{true}$

theorem NEW1_ vc_ fsb_ pre
$\forall NEW1FSBSig \mid true \bullet \mathbf{pre}\ NEW1$

theorem NEW1Exact_ vc_ fsb_ pre
$\forall NEW1ExactFSBSig \mid true \bullet \mathbf{pre}\ NEW1Exact$

theorem NEW1Bigger_ vc_ fsb_ pre
$\forall NEW1BiggerFSBSig \mid true \bullet \mathbf{pre}\ NEW1Bigger$

theorem NEW1Leo_ vc_ fsb_ pre
$\forall NEW1LeoFSBSig \mid true \bullet \mathbf{pre}\ NEW1Leo$

theorem dlDISPOSE1AbutLocJoins
$\forall DISPOSE1Join \bullet allLocs\,(abt2) = allLocs\,(join2)$

theorem dlDISPOSE1JoinAbuttNotEmpty
$$\forall DISPOSE1Join \bullet \forall r : Piece \mid r \in fr \land \neg\, r = p2 \land \neg\, p2 \in fr \land \neg\, p2\ \mathbf{abutt}\ r \bullet \neg\, abt2 = \{\} \Leftrightarrow (\forall k : Piece \mid k \in fr \bullet \neg\, p2\ \mathbf{abutt}\ k)$$

theorem dlDISPOSE1JoinValue
$\forall DISPOSE1Join \bullet join2 = \{p2\} \cup \{q : Piece \mid q \in fr \land p2\ \mathbf{abutt}\ q\}$

theorem dlDISPOSE1AbtValue
$\forall DISPOSE1Join \bullet abt2 = \{q : Piece \mid q \in fr \land p2\ \mathbf{abutt}\ q\}$

theorem dlDISPOSE1NP2LocValue
$\forall DISPOSE1Join \bullet np2.LOC = minLoc\ join2 \land np2.SIZE = sumSize\ join2$

theorem dlDISPOSE1FrValue
$\forall DISPOSE1Join \bullet fr \in Free1$

theorem dlDISPOSE1P2Value
$\forall DISPOSE1Join \bullet p2 \in Piece$

theorem dlDISPOSE1JoinNoAbuttV3
$$\forall DISPOSE1Join \bullet \forall r : Piece \mid r \in fr \land \neg\, r \in join2 \land \neg\, p2 \in fr \land \neg\, p2\ \mathbf{abutt}\ r \land locs\_of\ \ p2 \cap locs\ fr = \{\} \bullet join2 = \{p2\}$$

theorem rule lDISPOSE1FSBNewPieceLocIsNew
$\forall DISPOSE1Join \mid \neg\, p2 \in fr \bullet \neg\, np2.LOC \in allLocs\,(f \setminus join2)$

theorem rule lDISPOSE1FSBNonJoinPieceAfterNewPiece
$\forall DISPOSE1Join \bullet \forall r : Piece \mid r \in fr \land \neg r \in join2 \land$
$\neg p2 \in fr \land locs\_of\ p2 \cap locs\ fr = \{\} \land$
$np2.LOC < r.LOC \bullet np2$ **before** r

theorem rule lDISPOSE1FSBNonJoinPieceBeforeNewPiece
$\forall DISPOSE1Join \bullet \forall r : Piece \mid r \in fr \land \neg r \in join2 \land$
$\neg p2 \in fr \land locs\_of\ p2 \cap locs\ fr = \{\} \land$
$r.LOC < np2.LOC \bullet r$ **before** $np2$

theorem rule lDISPOSE1FSBAuxLemma
$\forall DISPOSE1Join \mid locs\_of\ p2 \cap locs\ fr = \{\} \bullet \{np2\} \cup (fr \setminus join2) \in Free1$

theorem DISPOSE1_ vc_ fsb_ pre
$\forall DISPOSE1FSBSig \mid true \bullet$ **pre** $DISPOSE1$

theorem DISPOSE1Leo_ vc_ fsb_ pre
$\forall DISPOSE1LeoFSBSig \mid true \bullet$ **pre** $DISPOSE1Leo$

4.6 Playfull lemmas

$NEW1DISPOSE1 \widehat{=} NEW1 \fatsemi DISPOSE1[r!/p?]$

$Init1NEW1DISPOSE1 \widehat{=} Init1 \fatsemi NEW1 \fatsemi DISPOSE1[r!/p?]$

Schema $NEW1DISPOSE1FSBSig$

$Heap1$
$s? : \mathbb{N}$

$\exists q : Piece \bullet q \in f1 \land q.SIZE \geq s?$

Schema $Init1NEW0DISPOSE1FSBSig$

$Init1[f1/f1']$
$memSize?, s? : \mathbb{N}$

$s? \leq memSize?$

Schema $NEW1DISPOSE1CancelsSig$

$Heap1$
$s? : \mathbb{N}$
$r! : Piece$

$\exists q : Piece \bullet q.SIZE = s? \land q = r! \land q \in f1$

$$
\begin{array}{|l}
\hline
\textit{Init1NEW1DISPOSE1CancelsSig} \\
\quad Heap1' \\
\quad memSize?, s? : \mathbb{N} \\
\quad r! : Piece \\
\hline
\quad s? = r!.SIZE \\
\quad r!.LOC + s? \leq memSize? \\
\hline
\end{array}
$$

theorem NEW1DISPOSE1_vc_fsb_pre
$\forall NEW1DISPOSE1FSBSig \bullet \mathbf{pre}\ NEW1DISPOSE1$

theorem Init1NEW1DISPOSE1_vc_fsb_pre
$\forall Init1NEW0DISPOSE1FSBSig \bullet \mathbf{pre}\ Init1NEW1DISPOSE1$

theorem NEW1DISPOSE1Cancels
$\forall NEW1DISPOSE1CancelsSig \bullet NEW1DISPOSE1 \Leftrightarrow \Xi Heap1$

theorem Init1NEW1DISPOSE1Cancels
$\forall Init1NEW1DISPOSE1CancelsSig \bullet Init1NEW1DISPOSE1 \Leftrightarrow Init1$

4.7 Refinement theorems

memSize? input *must* be greater than zero, otherwise the *Free*1 invariant won't hold

theorem Heap1_ vc_ ref_ fs_ init
$\forall Init1 \bullet \exists RetrFree0Free1' \mid true \bullet Init0$

theorem NEW1_ vc_ ref_ fs_ applic
$\forall NEW0FSBSig;\ NEW1FSBSig;\ RetrFree0Free1 \mid \mathbf{pre}\ NEW0 \bullet \mathbf{pre}\ NEW1$

theorem NEW1_ vc_ ref_ fs_ correct
$\forall Heap0;\ NEW1;\ RetrFree0Free1 \mid \mathbf{pre}\ NEW0 \bullet \exists Heap0' \mid RetrFree0Free1' \bullet NEW0$

theorem DISPOSE1_ vc_ ref_ fs_ applic
$\forall DISPOSE0FSBSig;\ DISPOSE1FSBSig;\ RetrFree0Free1 \mid \mathbf{pre}\ DISPOSE0 \bullet \mathbf{pre}\ DISPOSE1$

The real theorem behind this is the one that ensures np does the joining job properly. That is, makes sure that the ranges are bounded by np. I use the subsetRange correspondence here for the shape of the lemma. Because of our use of DISPOSE1Join, I will make it frules. These depend on an inductive proof about *minLoc*/*sumSize*.

theorem OLD-IGNORED frule flDISPOSE1JoinCorrectnessNPMinLocLemma
$\forall DISPOSE1Join \bullet np2.LOC \leq p2.LOC$

theorem OLD-IGNORED rule flDISPOSE1JoinCorrectnessNPBoundedSizeLemma
$\forall DISPOSE1Join \bullet p2.LOC + p2.SIZE \leq np2.LOC + np2.SIZE$

theorem DISPOSE1_ vc_ ref_ fs_ correct
$\forall Heap0;\ DISPOSE1;\ RetrFree0Free1 \mid \mathbf{pre}\ DISPOSE0 \bullet$
$\quad \exists Heap0' \mid RetrFree0Free1' \bullet DISPOSE0$

Declarations	**This Chapter**	**Globally**
Unboxed items	16	32
Axiomatic definitions	4	5
Generic axiomatic defs.	0	1
Schemas	24	41
Generic schemas	0	0
Theorems	90	148
Proofs	0	0
Total	**134**	**227**

Table 4.1: Summary of Z declarations for Chapter 4.

Declarations	**This Chapter**	**Globally**
Unboxed items	16	32
Axiomatic definitions	4	5
Generic axiomatic defs.	0	1
Schemas	24	41
Generic schemas	0	0
Theorems	90	148
Proofs	0	0
Total	**134**	**227**

Table 4.2: Summary of Z declarations for Chapter 4.

Chapter 5

Proofs

section *arithmeticProofs* **parents** *arithmetic*

5.1 Arithmetics Proofs

Proofs are trivial, as expected

proof[*lLessNeg*]
simplify;
∎

proof[*lGreaterNeg*]
simplify;
∎

proof[*lLeqNeg*]
simplify;
∎

proof[*lGeqNeg*]
simplify;
∎

proof[*lLessFlip*]
simplify;
∎

proof[*lGreaterFlip*]
simplify;
∎

proof[*lLeqFlip*]
simplify;
∎

proof[*lGeqFlip*]
simplify;
■

proof[*lLessPromote*]
simplify;
■

proof[*lGreaterPromote*]
simplify;
■

section *setsProofs* **parents** *sets*

5.2 Set Proofs

5.2.1 Bigcup proofs

proof[*dlBigCupAsBigU*]
apply extensionality;
prove;
apply dBigU to expression $bigU[XX]\ SS$;
prove;
cases;
apply inBigcup to predicate $x \in \bigcup[XX]\ SS$;
prove;
split $x \in XX$;
rewrite;
cases;
instantiate $S == B$;
prove;
next;
rearrange;
split $\exists\ S\_\_0 : \mathbb{P}\ XX \bullet S\_\_0 \in SS \land x \in S\_\_0$;
simplify;
prove;
next;
instantiate $B == S$;
prove;
next;
■

proof[*dlInBigU*]
  split $x \in bigU[XX]\, SS$;
  cases;
  rewrite;
  apply dBigU;
  prove;
  instantiate $ss == S$;
  rewrite;
  next;
  rewrite;
  rearrange;
  split $(\exists ss : SS \bullet x \in ss)$;
  rewrite;
  apply dBigU;
  prove;
  instantiate $S == ss$;
  prove;
  next;
  ■

Andrius: how does CZT parses this first proof command?

proof[*dlInPowerBigU*]
    apply inPower to predicate $x \in \mathbb{P}\,(bigU\ [XX]\ SS)$;
    apply dBigU;
    prove;
    instantiate $S == x$;
    prove;
  ■

proof[*lCupDiffSubsumption*]
  apply extensionality;
    with normalization prove;
  ■

proof[*lDiffElimOverNonOverlappingSets*]
  apply extensionality;
    prove;
    instantiate $x == y$;
    prove;
  ■

5.2.2 Range proofs

proof[*dlRangeCapLeft*]
  apply extensionality;
  prove;
  ■

proof[*dlRangeCapRight*]
  apply extensionality;
  prove;
  ■

proof[*dlRangeCapEmpty*]
split $B < A$;
prove;
split $D < C$;
prove;
split $B < C$;
cases;
apply dlRangeCapLeft;
simplify;
apply rangeNull;
simplify;
next;
apply dlRangeCapRight;
simplify;
apply rangeNull;
simplify;
next;
■

proof[*dlRangeSumSubset*]
prove;
■

proof[*dlRangeDifference*]
apply extensionality;
prove;
■

proof[*dlRangeSubsumes*]
prove;
■

5.2.3 Minimum proofs

proof[*lMinWithinSubset*]
use finiteSetHasMin;
use minProperty;
prove;
■

5.2.4 Finiteness proofs

proof[*lCrossFinite*2]
rewrite;
■

proof[*lFinsetSubset*]
prove by reduce;
■

proof[*lFinsetSubsetKnown*]
prove by reduce;
∎

proof[*lIsFinite*]
prove;
∎

proof[*lBijectionFinite*]
use functionFinite$[X, Y][A := A, B := B, f := f]$;
use finiteFunction$[X, Y][f := f]$;
use functionFinite$[X, Y][A := X, B := Y, f := f]$;
use finiteFunction$[Y, X][f := (\_ \sim)[X, Y](f)]$;
use functionFinite$[Y, X][A := B, B := A, f := (\_ \sim)[X, Y](f)]$;
prove by rewrite;
∎

proof[*lNonMaximalCardEquiv*]
use sizeDef$[X][S := S]$;
use sizeDef$[A][S := S]$;
prove;
use lBijectionFinite$[\mathbb{Z}, X][f := f, A := 1 \mathrel{..} (\# \_)[A](S), B := S]$;
use lBijectionFinite$[\mathbb{Z}, X][f := f\_\_0, A := 1 \mathrel{..} (\# \_)[X](S), B := S]$;
with disabled (*sizeRange*) *prove*;
use cardIsNonNegative$[A][S := S]$;
prove;
∎

proof[*lNonMaxDomEquiv*]
apply extensionality to predicate $\mathrm{dom}\,[X, Y]\, R = \mathrm{dom}\,[A, B]\, R$;
invoke $(\_ \leftrightarrow \_)$;
with enabled (*inDom*) *prove*;
apply inPower to predicate $R \in \mathbb{P}\,(A \times B)$;
cases;
instantiate $y\_\_2 == y\_\_0$;
instantiate $e == (x, y)$;
prove;
next;
instantiate $y\_\_3 == y\_\_1$;
instantiate $e == (y\_\_0, y)$;
prove;
next;
∎

proof[*lNonMaxRanEquiv*]
apply extensionality ;
invoke $(\_ \leftrightarrow \_)$;
with enabled (*inRan*) *prove*;
apply inPower to predicate $R \in \mathbb{P}(A \times B)$;
cases;
instantiate $x\_\_2 == x\_\_0$;
instantiate $e == (x, x\_\_0)$;
prove;
next ;
instantiate $x\_\_3 == x\_\_1$;
instantiate $e == (x, y)$;
prove;
next ;
■

proof[*lSeqFinite*]
use lFinsetSubsetKnown$[Z := \mathbb{Z} \times X, Y := (\mathbb{N} \times X), X := s]$;
rearrange;
rewrite;
■

proof[*lRanSeqFinite*]
use lNonMaxRanEquiv$[\mathbb{Z}, X][A := \mathbb{N}_1, B := A, R := s]$;
rearrange;
rewrite;
equality substitute $\mathrm{ran}[\mathbb{Z}, X]\, s$;
use seq_type$[A]$;
apply inPower to predicate $\mathrm{seq}\ A \in \mathbb{P}(\mathbb{N}_1 \twoheadrightarrow\!\!\!\!\!\!\!+\ A)$;
instantiate $e == s$;
rearrange;
simplify;
use finiteFunction$[\mathbb{N}_1, A][f := s]$;
rearrange;
simplify;
■

section *Heap2CBJ0Proofs* **parents** *Heap2CBJ0*

5.3 CBJ2 Heap0 Proofs

5.3.1 WD Proofs

proof[*locsOf*\$*domainCheck*]
with enabled (*Loc*) *prove by reduce*;
■

proof[*NEW0*\$*domainCheck*]
with enabled (*Loc*) *prove by reduce*;
■

proof[$DISPOSE0\$domainCheck$]
with enabled (Loc) *prove by reduce*;
∎

proof[$NEW0FSBSig\$domainCheck$]
with enabled (Loc) *prove by reduce*;
∎

proof[$DISPOSE0FSBSig\$domainCheck$]
with enabled (Loc) *prove by reduce*;
∎

proof[$Loc\_vc\_fsb\_horiz\_def$]
instantiate $Loc == \{0\}$;
prove;
∎

proof[$Free0\_vc\_fsb\_horiz\_def$]
instantiate $Free0 == \emptyset$;
prove;
∎

5.3.2 Feasibility Proofs

proof[$Piece\_vc\_fsb\_state$]
instantiate $LOC == 0$, $SIZE == 1$;
with enabled (Loc) *prove by reduce*;
∎

proof[$Heap0\_vc\_fsb\_state$]
instantiate $f0 == \emptyset$;
prove by reduce;
∎

proof[$Init0\_vc\_fsb\_init$]
prove by reduce;
∎

proof[$NEW0\_vc\_fsb\_pre$]
prove by reduce;
instantiate $r! == q$;
prove;
∎

proof[$DISPOSE0\_vc\_fsb\_pre$]
prove by reduce;
∎

5.3.3 Lemmas proofs

proof[*gLocMaxType*]
with enabled (*Loc*) *prove by reduce*;
■

proof[*gLocType*]
with enabled (*Loc*) *prove by reduce*;
■

proof[*fPieceLOCMaxType*]
with enabled (*Piece$member*) *prove by reduce*;
■

proof[*fPieceSIZEMaxType*]
with enabled (*Piece$member*) *prove by reduce*;
■

proof[*fPieceSIZEIsNat*1]
with enabled (*Piece$member*) *prove by reduce*;
■

proof[*fPieceLOCIsNat*]
with enabled (*Piece$member*, *Loc*) *prove by reduce*;
■

proof[*gPieceMaxType*]
prove;
■

proof[*lLocsOfResMaxType*]
apply dlLocsOfDef;
prove;
■

proof[*gLocsOfRelType*]
use locsOf$declaration;
invoke $(\_ \rightarrow \_)$;
invoke $(\_ \nrightarrow \_)$;
invoke $(\_ \leftrightarrow \_)$;
rewrite;
trivial rewrite;
prenex;
apply inPower;
prenex;
instantiate $e\_\_0 \;==\; e$;
*apply inCross*2;
with enabled (*Loc*) *prove by reduce*;
■

THIS IS RIDICULOUS! THERE IS SOME MISSING TYPE BRIDGE

proof[*lLocsOfIsTotal*]
use locsOf\$*declaration*;
invoke $(\_ \rightarrow \_)$;
apply inDom;
rewrite;
instantiate $x == p$;
prove;
instantiate $y\_\_1 == y$;
with enabled (*Loc*) *prove by reduce*;
∎

proof[*dlLocsOfProp*]
apply dlLocsOfDef;
prove;
∎

5.3.4 Playfull lemmas Proofs

proof[*dlArithAux*]
prove;
∎

proof[*NEW0DISPOSE0_vc_fsb_pre*]
prove by reduce;
instantiate $r! == q$;
prove by reduce;
apply extensionality to predicate $locs\_of\ q \cap (f0 \setminus locs\_of\ q) = \{\}$;
prove;
∎

proof[*Init0NEW0DISPOSE0_vc_fsb_pre*]
prove by reduce;
instantiate $r! == \theta\ Piece[LOC := 0,\ SIZE := s?]$;
apply dlLocsOfProp;
prove by reduce;
instantiate $p == \theta\ Piece[LOC := 0,\ SIZE := s?]$;
prove by reduce;
apply extensionality;
prenex;
rewrite;
∎

proof$[NEW0DISPOSE0Cancels]$
$split\ \Xi\ Heap0;$
$rewrite;$
$cases;$
$prove\ by\ reduce;$
$equality\ substitute\ f0';$
$cases;$
$apply\ extensionality\ to\ predicate\ f0\ =\ locs\_of\ \ r!\ \cup\ (f0\ \setminus\ locs\_of\ \ r!);$
$with\ normalization\ prove;$
$next;$
$instantiate\ p\ \ ==\ \ r!;$
$rewrite;$
$next;$
$apply\ extensionality\ to\ predicate\ \ locs\_of\ \ r!\ \cap\ (f0\ \setminus\ locs\_of\ \ r!)\ =\ \{\};$
$prove;$
$next;$
$rearrange;$
$split\ NEW0DISPOSE0;$
$rewrite;$
$prove\ by\ reduce;$
$apply\ extensionality\ to\ predicate\ f0\ =\ locs\_of\ \ r!\ \cup\ (f0\ \setminus\ locs\_of\ \ r!);$
$with\ normalization\ prove;$
$next;$
$\blacksquare$

proof[$Init0NEW0DISPOSE0Cancels$]
*split Init*0;
rewrite;
cases;
with disabled ($Init0NEW0DISPOSE0CancelsSig$) *reduce*;
instantiate p == $r!$;
rearrange;
rewrite;
cases;
prove by reduce;
next;
apply extensionality;
with disabled (*inRange*) *prove*;
cases;
rearrange;
instantiate $x\_\_0$ == x;
rearrange;
simplify;
next;
split $y \in locs\_of\ r!$;
simplify;
cases;
apply dlLocsOfProp;
with disabled (*inRange*) *with enabled* (*Piece$member*, *Loc*) *prove by reduce*;
instantiate $y\_\_0$ == y;
with disabled (*inRange*) *with normalization rewrite*;
next;
instantiate $y\_\_1$ == y;
prove;
next;
apply extensionality;
with disabled (*inRange*) *prove*;
next;
instantiate p == θ *Piece*[LOC := $r!.LOC$, $SIZE$:= $s?$];
apply dlLocsOfProp;
with enabled (*Piece$member*, *Loc*) *prove by reduce*;
next;
rearrange;
split $Init0NEW0DISPOSE0$;
rewrite;
*invoke Init*0;
rewrite;
apply extensionality;
invoke;
prenex;
equality substitute;
rearrange;
cases;
with normalization rewrite;
apply dlLocsOfProp to expression $locs\_of\ r!$;
rewrite;
next;
rewrite;
next;
■

proof[$NEW0DISPOSE0FSBSig\$domainCheck$]
prove;
■

proof[$NEW0DISPOSE0CancelsSig\$domainCheck$]
prove;
■

5.3.5 CWG0 original proofs

proof[$hasSeq\$domainCheck$]
prove by reduce;
■

proof[$NEWCWG0\_vc\_fsb\_pre$]
prove by reduce;
instantiate $s == t$;
prove;
instantiate $i\_\_0 == i,\ j\_\_0 == j$;
prove;
■

proof[$DISPOSECWG0\_vc\_fsb\_pre$]
prove by reduce;
■

proof[$NEWCWG0DISPOSECWG0Cancels$]

split Ξ *Heap*0;
rewrite;
cases;
invoke $NEWCWG0DISPOSECWG0$;
invoke $NEWCWG0DISPOSECWG0CancelsSig$;
rewrite;
invoke $NEWCWG0$;
invoke $DISPOSECWG0$;
invoke Ξ *Heap*0;
invoke Δ *Heap*0;
invoke *Heap*0;
invoke *Free*0;
rewrite;
prenex;
rearrange;
instantiate $s == t$;
rewrite;
equality substitute $f0'$;
invoke (**hasSeq** _);
rewrite;
cases;
apply extensionality to predicate $f0 = r! \cup (f0 \setminus r!)$;
prenex;
with normalization rewrite;
next;
apply extensionality to predicate $r! \cap (f0 \setminus r!) = \{\}$;
prove;
next;
rearrange;
split $NEWCWG0DISPOSECWG0$;
rewrite;
prove by reduce;
apply extensionality to predicate $f0 = \mathrm{ran}\ s \cup (f0 \setminus \mathrm{ran}\ s)$;
with normalization prove;
next;
■

5.4 Commented out proofs

For knowing how to translate to goodmunds strategy language

```
begin{zproof}[NEW0\_ vc\_ fsb\_ pre]
% expand / simplify / rearrange = STR1(goal)
invoke NEW0; % expand defs in the goal
simplify;    %
rearrange;
% expand / simplify / rearrange = STR1(hyp)
invoke NEW0FSBSig;
simplify;
% expand / simplify / rearrange = STR1(goal)
invoke \Delta Heap0;
simplify;
rearrange;
% expand / simplify / rearrange = STR1(hyp)
invoke Heap0;
% instantiate goal: pick witness / simplify [failed] = STR2(goal)
instantiate r! == \theta Piece[LOC := 0, SIZE := s?];
rewrite;
% STR1(goal)
invoke Piece;
rewrite;
```

```
invoke Free0;
invoke Loc;
rewrite;
apply inPower to predicate f0 \setminus \locsOf(\theta Piece [LOC := 0, SIZE:= s?]) \in \power \nat;
prenex;
rewrite;
rearrange;
% eliminate quantifiers (goal, quantifiers)
prenex;
end{zproof}
```

section *Heap2CBJ1Proofs* **parents** *Heap2CBJ1*

5.5 CBJ2 Heap1 Proofs

5.5.1 WD Proofs

proof[*locs$domainCheck*]
with enabled (*Loc*) *prove by reduce*;
■

proof[*allLocs$domainCheck*]
with enabled (*Loc*) *prove by reduce*;
■

proof[*minLoc$domainCheck*]
with enabled (*Loc*) *prove by reduce*;
apply dlAllLocs;
rewrite;
apply extensionality to predicate $\{p : Piece \mid p \in ps \bullet p\ .\ LOC\} = \{\}$;
apply extensionality to predicate $ps = \{\}$;
prove;
instantiate $x\_\_0 == x.LOC$;
rewrite;
instantiate $p == x$;
rewrite;
■

proof[*sumSize$domainCheck*]
with enabled (*Loc*) *prove by reduce*;
■

proof[*DISPOSE1$domainCheck*]
with enabled (*Loc*) *prove by reduce*;
■

proof[*DISPOSE1Leo$domainCheck*]
with enabled (*Loc*) *prove by reduce*;
■

proof[$DISPOSE1FSBSig\$domainCheck$]
with enabled (Loc) *prove by reduce*;
■

proof[$DISPOSE1LeoFSBSig\$domainCheck$]
with enabled (Loc) *prove by reduce*;
■

proof[$RetrFree0Free1\$domainCheck$]
with enabled (Loc) *prove by reduce*;
■

proof[$Free1\_vc\_fsb\_horiz\_def$]
instantiate $Free1 == \{\}$;
prove;
■

5.5.2 Feasibility Proofs

proof[$Heap1\_ vc\_ fsb\_ state$]
instantiate $f1 == \varnothing$;
with enabled ($Free1$, sep _, **unique** _) *prove by reduce*;
■

proof[$Init1\_ vc\_ fsb\_ init$]
instantiate $memSize\_\_0? == memSize?, f1' == \{ \theta\ Piece[LOC := 0, SIZE := memSize?] \}$;
prove by reduce;
apply $dlInFree1$;
apply $dlInInvFree1Sep$;
apply $dlInInvFree1Unique$;
prove by reduce;
■

proof[$NEW1\_vc\_fsb\_pre$]
invoke $NEW1FSBSig$;
prenex;
apply $dlNEW1Equiv$;
split $q.SIZE = s?$;
rewrite;
cases;
prove by reduce;
instantiate $t == q,\ r! == q,\ f1' == f1 \setminus \{ q \}$;
prove;
with enabled ($Piece\$member$) *prove*;
next;
with disabled ($NEW1Exact$) *prove by reduce*;
instantiate $t == q,\ r! == \theta\ Piece[LOC := q.LOC,\ SIZE := s?]$,
$f1' == f1 \setminus \{ q \} \cup \{ \theta\ Piece[LOC := q.LOC + s?,\ SIZE := q.SIZE - s?] \}$;
with disabled ($NEW1Exact$) *prove by reduce*;
apply $lFree1UnitUnionInType$;
with disabled ($NEW1Exact$) *reduce*;
prenex;
conjunctive;
rewrite;
cases;
apply $dlAllLocs$;
rewrite;
apply $dlInFree1$ *to predicate* $f1 \in Free1$;
apply $dlInInvFree1Sep$;
prenex;
instantiate $p1 == q,\ p2 == p$;
prove;
apply $dlPieceBefore$;
rewrite;
next;
apply $dlInFree1$ *to predicate* $f1 \in Free1$;
apply $dlInInvFree1Sep$;
apply $dlPieceBefore$;
reduce;
instantiate $p1 == q,\ p2 == r$;
prove;
next;
apply $dlInFree1$ *to predicate* $f1 \in Free1$;
apply $dlInInvFree1Sep$;
instantiate $p1 == r,\ p2 == q$;
rearrange;
rewrite;
split $r\ .\ LOC < q\ .\ LOC$;
rewrite;
cases;
apply $dlPieceBefore$;
reduce;
next;
rearrange;
split $r.LOC = q.LOC$;
cases;
apply $dlInInvFree1Unique$;
rearrange;
instantiate $p1 == r,\ p2 == q$;
prove;
next;
rearrange;
simplify;
instantiate $p1 == q,\ p2 == r$;
prove;
apply $dlPieceBefore$;

proof[$DISPOSE1\_vc\_fsb\_pre$]
$apply\ dlDISPOSE1Equiv;$
$prove\ by\ reduce;$
$use\ dlDISPOSE1JoinWitness;$
$invoke\ Heap1;$
$prenex;$
$instantiate\ abt2\_\_0\ ==\ abt2,\ join2\_\_0\ ==\ join2,\ np2\_\_0\ ==\ np2;$
$prove;$
■

used lDISPOSE1FSBAuxLemma

proof[$lDISPOSE1FSBNonJoinPieceAfterNewPiece$]
$use\ dlDISPOSE1JoinNoAbuttV3;$
$rearrange;$
$simplify;$
$invoke\ DISPOSE1Join;$
$rearrange;$
$split\ (p2\ \mathbf{abutt}\ r);$
$cases;$
$prove;$
$next;$
$simplify;$
$rearrange;$
$split\ join2 = \{p2\} \land np2 = \theta\ (Piece\ [LOC := minLoc\ join2,\ SIZE := sumSize\ join2]);$
$rewrite;$
$equality\ substitute\ join2;$
$equality\ substitute\ np2;$
$apply\ dlPieceBefore;$
$apply\ dlLocsCapEmpty;$
$rewrite;$
$apply\ dlInAbutt\ to\ predicate\ p2\ \mathbf{abutt}\ r;$
$apply\ dlInFuse;$
$rewrite;$
$instantiate\ t\ ==\ r;$
$apply\ extensionality\ to\ predicate\ locs\_of\ p2 \cap locs\_of\ r = \{\};$
$rewrite;$
$instantiate\ x\ ==\ r.LOC;$
$rewrite;$
$apply\ dlLocsOfProp\ to\ expression\ locs\_of\ p2;$
$apply\ dlLocsOfProp\ to\ expression\ locs\_of\ r;$
$rearrange;$
$rewrite;$
$next;$
■

proof[$lDISPOSE1FSBNonJoinPieceBeforeNewPiece$]
use $dlDISPOSE1JoinNoAbuttV3$;
rearrange;
simplify;
invoke $DISPOSE1Join$;
rearrange;
split ($p2$ **abutt** r);
cases;
prove;
next;
simplify;
rearrange;
split $join2 = \{p2\} \land np2 = \theta\,(Piece\,[LOC := minLoc\,join2,\ SIZE := sumSize\,join2])$;
rewrite;
equality substitute $join2$;
equality substitute $np2$;
apply $dlPieceBefore$;
apply $dlLocsCapEmpty$;
rewrite;
apply $dlInAbutt$ *to predicate* $p2$ **abutt** r;
rewrite;
instantiate $t == r$;
rewrite;
apply $extensionality$ *to predicate* $locs\_of\ p2 \cap locs\_of\ r = \{\}$;
rewrite;
instantiate $x == p2.LOC$;
rewrite;
apply $dlLocsOfProp$ *to expression* $locs\_of\ p2$;
apply $dlLocsOfProp$ *to expression* $locs\_of\ r$;
rearrange;
rewrite;
apply $lLeqNeg$;
rewrite;
apply $lGreaterFlip$;
rewrite;
apply $dlInFuse$ *to predicate* r **fuse** $p2$;
rewrite;
next;
■

proof[$lDISPOSE1FSBAuxLemma$]
$split\ p2\ \in\ fr$;
$cases$;
$apply\ dlLocsCapEmpty$;
$rewrite$;
$instantiate\ t\ ==\ p2$;
$rewrite$;
$apply\ capSubsetRight$;
$rewrite$;
$apply\ extensionality$;
$apply\ dlLocsOfProp$;
$prove$;
$instantiate\ x\ ==\ p2.LOC$;
$prove$;
$next$;
$apply\ cupCommutes$;
$with\ disabled\ (cupCommutes)\ rewrite$;
$next$;
■

proof[$DISPOSE1Join\_\ vc\_\ fsb\_\ state$]
$split\ \neg\ \exists\ p :\ Piece\ \bullet\ true$;
$cases$;
$instantiate\ p\ ==\ \theta\ Piece[LOC\ :=\ 0,\ SIZE\ :=\ 1]$;
$with\ enabled\ (Piece\$member,\ Loc)\ prove\ by\ reduce$;
$next$;
$prenex$;
$instantiate\ fr\ ==\ \{\},\ abt2\ ==\ \{\},$
$join2\ ==\ \{\ p\ \},\ p2\ ==\ p,\ np2\ ==\ \theta\ Piece[LOC\ :=\ minLoc(\{p\}),\ SIZE\ :=\ sumSize(\{p\})]$;
$invoke\ DISPOSE1JoinFSBSig$;
$simplify$;
$invoke\ DISPOSE1Join$;
$reduce$;
$apply\ extensionality$;
$prove$;
$next$;
■

proof[$RetrFree0Free1\_\ vc\_\ fsb\_\ state$]
$instantiate\ f0\ ==\ \{\},\ f1\ ==\ \{\}$;
$prove\ by\ reduce$;
$apply\ dlLocsDef$;
$rewrite$;
$apply\ extensionality$;
$prove$;
$apply\ dlInBigU$;
$prove$;
■

proof[$NEW1Bigger\_vc\_fsb\_pre$]
prove by reduce;
instantiate $t == q$;
prove;
apply lFree1UnitUnionInType;
reduce;
prenex;
conjunctive;
rewrite;
cases;
apply dlAllLocs;
rewrite;
apply dlInFree1 to predicate $f1 \in Free1$;
apply dlInInvFree1Sep;
prenex;
instantiate $p1 == q, p2 == p$;
prove;
apply dlPieceBefore;
rewrite;
next;
apply dlInFree1 to predicate $f1 \in Free1$;
apply dlInInvFree1Sep;
apply dlPieceBefore;
reduce;
instantiate $p1 == q, p2 == r$;
prove;
next;
apply dlInFree1 to predicate $f1 \in Free1$;
apply dlInInvFree1Sep;
instantiate $p1 == r, p2 == q$;
rearrange;
rewrite;
split $r . LOC < q . LOC$;
rewrite;
cases;
apply dlPieceBefore;
reduce;
next;
rearrange;
split $r.LOC = q.LOC$;
cases;
apply dlInInvFree1Unique;
rearrange;
instantiate $p1 == r, p2 == q$;
prove;
next;
rearrange;
simplify;
instantiate $p1 == q, p2 == r$;
prove;
apply dlPieceBefore;
rewrite;
next;
∎

proof[$NEW1Exact\_vc\_fsb\_pre$]
prove by reduce;
instantiate $t == q$;
with enabled ($Piece\$member$) *prove*;
∎

NOTE: quite neat example of proof refactoring here for NEW1Exact/Bigger

proof[$NEW1Leo\_ vc\_ fsb\_ pre$]
use $NEW1\_vc\_fsb\_pre$;
rearrange;
invoke $NEW1FSBSig$;
invoke $NEW1LeoFSBSig$;
simplify;
prenex;
apply $dlNEW1Equiv$;
instantiate $f1\_\_0' == f1'$;
instantiate $r\_\_0! == r!$;
prove;
∎

proof[$DISPOSE1Leo\_ vc\_ fsb\_ pre$]
use $DISPOSE1\_vc\_fsb\_pre$;
rearrange;
invoke $DISPOSE1FSBSig$;
invoke $DISPOSE1LeoFSBSig$;
prenex;
rewrite;
use $dlDISPOSE1Equiv$;
instantiate $f1\_\_0' == f1'$;
rewrite;
∎

5.5.3 Lemmas proofs

proof[$fFree1ElemMaxType$]
invoke $Free1$;
prove;
∎

proof[$fFree1ElemType$]
invoke $Free1$;
prove;
∎

proof[$fFree1ElemFinType$]
invoke $Free1$;
prove;
∎

proof[$gLocsRelType$]
prove;
∎

proof[*lLocsIsTotal*]
prove;
∎

proof[*gAllLocsRelType*]
prove;
∎

proof[*lAllLocsIsTotal*]
prove;
∎

proof[*gMinLocRelType*]
prove;
∎

proof[*lMinLocIsTotal*]
prove;
∎

proof[*lMinLocResIsNat*]
use applyInRanFun[$\mathbb{P}_1$ *Piece*, *Loc*][f := *minLoc*, a := f];
rearrange;
simplify;
invoke Loc;
rewrite;
∎

proof[*gSumSizeRelType*]
prove;
∎

proof[*lSumSizeIsTotal*]
prove;
∎

proof[*lSumSizeResIsNat*]
use applyInRanFun[$\mathbb{F}$ *Piece*, $\mathbb{N}$][f := *sumSize*, a := f];
rearrange;
simplify;
apply inNat;
rewrite;
∎

proof[*lSumSizeResIsNat*1]
apply extensionality;
prove;
split $\neg\ \exists\ v : \mathbb{F}\ Piece;\ \ p : \ Piece\ \bullet\ f\ =\ \{p\}\ \cup\ v$;
cases;
instantiate $v\ ==\ f\ \setminus\ \{x\},\ p\ ==\ x$;
prove;
next;
prove;
apply dlSumSizeInduct;
prove;
use lSumSizeResIsNat$[f\ :=\ v]$;
rearrange;
simplify;
next;
■

proof[*dlInFree*1]
*invoke Free*1;
prove;
■

proof[*dlInInvFree*1*Sep*]
invoke (*sep* _);
prove;
split $(\forall\ p1 :\ fr;\ \ p2 :\ fr\ \mid\ p1.LOC\ <\ p2.LOC\ \bullet\ p1$ **before** $p2)$;
rewrite;
cases;
prove;
instantiate $p1\_\_0 == p1,\ p2\_\_0 == p2$;
prove;
next;
prove;
instantiate $p1\_\_0 == p1,\ p2\_\_0 == p2$;
prove;
next;
■

proof[*dlInInvFree*1*Unique*]
invoke (**unique** _);
prove;
split $(\forall\ p1 :\ fr;\ \ p2 :\ fr\ \mid\ p1.LOC\ =\ p2.LOC\ \bullet\ p1\ =\ p2)$;
rewrite;
cases;
prove;
instantiate $p1\_\_0 == p1,\ p2\_\_0 == p2$;
prove;
next;
prove;
instantiate $p1\_\_0 == p1,\ p2\_\_0 == p2$;
prove;
next;
■

proof[*lFree1Empty*]
apply dlInFree1;
apply dlInInvFree1Sep;
apply dlInInvFree1Unique;
prove;
■

proof[*dlPieceBefore*]
with enabled (_ **before** _) *prove by reduce*;
■

proof[*dlPieceWellPlaced*]
with enabled (_ **wellplaced** _, *dlPieceBefore*) *prove by reduce*;
apply dlInInvFree1Unique;
prove;
with normalization rewrite;
■

proof[*dlPieceExcludedMiddle*]
with enabled (*dlPieceBefore*, *Piece$member*, *Loc*) *prove by reduce*;
■

proof[*lPieceNotBeforeItself*]
apply dlPieceBefore;
prove;
■

proof[*dlFree1UnitUnionUniqueProp*]
split **unique** $(f \cup \{p\})$;
rewrite;
cases;
*apply dlInFree*1;
*apply dlInInvFree*1*Unique to predicate* **unique** $(f \cup \{p\})$;
prove;
instantiate $p1 == q,\ p2 == p$;
with normalization rewrite;
next;
rearrange;
simplify;
split $(\forall\ q : Piece \mid q \in f \land q.LOC = p.LOC \bullet q.SIZE = p.SIZE)$;
rewrite;
rearrange;
*apply dlInInvFree*1*Unique to predicate* **unique** $(f \cup \{p\})$;
cases;
prove;
next;
prenex;
*apply dlInFree*1;
*apply dlInInvFree*1*Unique*;
instantiate $p1\_0 == p1,\ p2\_0 == p2$;
prove;
instantiate $q == p1$;
instantiate $q == p2$;
with enabled (*Piece$member*) *prove*;
with normalization rewrite;
next;
■

proof[*lFree1UnitUnionUnique*]
*apply dlInFree*1 *to predicate* $f \cup \{p\} \in Free1$;
with enabled (*dlFree1UnitUnionUniqueProp*) *rewrite*;
apply dlAllLocs;
rewrite;
prenex;
instantiate $p\_0 == q$;
prove;
■

proof[*lFree1UnitUnionInType*]
$\quad$ *apply dlInFree1 to predicate* $f \cup \{t\} \in Free1$;
$\quad$ *rewrite*;
$\quad$ *apply dlInFree1*;
$\quad$ *apply dlInInvFree1Sep to predicate* $sep(f \cup \{t\})$;
$\quad$ *prove*;
$\quad$ *conjunctive*;
$\quad$ *rewrite*;
$\quad$ *cases*;
$\quad$ *apply dlInInvFree1Sep*;
$\quad$ *instantiate* $p1\_\_0 == p1,\ p2\_\_0 == p2$;
$\quad$ *prove*;
$\quad$ *next*;
$\quad$ *simplify*;
$\quad$ *instantiate* $r == p1$;
$\quad$ *prove*;
$\quad$ *next*;
$\quad$ *rearrange*;
$\quad$ *simplify*;
$\quad$ *instantiate* $r == p2$;
$\quad$ *prove*;
$\quad$ *next*;
$\quad$ ■

Needs extra conditions from lFree1UnitUnionInType side conditions

$\quad$ *ulLocsOfDisjFreeInvProp1*
$$\forall f : Free1;\ p1, p2 : Piece \mid p1 \in f \land \neg\, p2 \in f \bullet$$
$$\quad locs\_of\ \ p1 \cap locs\_of\ \ p2 = \{\}$$

$\quad$ *ulLocsOfDisjFreeInvProp2*
$$\forall f : Free1;\ p1, p2 : Piece \mid p1 \in f \land \neg\, p2 \in f \bullet$$
$$\quad \neg\, p1.LOC + p1.SIZE = p2.LOC$$

$\quad$ *ulLocsOfDisjFreeInvProp3*
$$\forall f : Free1;\ p1, p2 : Piece \mid p1 \in f \land \neg\, p2 \in f \bullet$$
$$\quad \neg\, p2.LOC + p2.SIZE = p1.LOC$$

Not needed anymore, but it's a nice lemma

proof[*dlLocsOfCapEmpty*]
$\quad$ *apply dlLocsOfProp*;
$\quad$ *rewrite*;
$\quad$ *apply dlRangeCapEmpty*;
$\quad$ *rewrite*;
$\quad$ *with normalization prove*;
$\quad$ ■

proof[$lFree1ReductionInType$]

apply $dlInFree1$;
prove;
cases;
apply $dlInInvFree1Sep$;
prove;
instantiate $p1\_\_0 == p1,\ p2\_\_0 == p2$;
prove;
next;
  apply $dlInInvFree1Unique$;
  prove;
instantiate $p1\_\_0 == p1,\ p2\_\_0 == p2$;
prove;
next;

■

proof[$fPieceLOCType$]

with enabled ($Piece\$member$) *prove by reduce*;

■

proof[*lLocsDistUnitDiff*]

apply dlLocsDef ;
prove;
apply extensionality;
prove;
*apply dlInFree*1;
apply dlInBigU;
prove;
cases;
cases;
apply dlLocsOfProp;
apply inPower;
instantiate e == p;
with enabled (*Piece*\$*member*) *prove*;
with normalization rewrite;
cases;
*apply dlInInvFree*1*Unique*;
instantiate $p1$ == p__0, $p2$ == p;
prove;
next;
*apply dlInInvFree*1*Sep*;
instantiate $p1$ == p__0, $p2$ == p;
instantiate $p1$ == p, $p2$ == p__0;
apply dlPieceBefore;
prove;
next;
instantiate ss__1 == ss;
prove;
instantiate p__1 == p__0;
prove;
next;
instantiate ss__2 == ss__0;
rewrite;
instantiate p__1 == p__0;
rewrite;
next;

∎

proof[$lLocsDistUnitCup$]

$apply\ dlLocsDef;$
$rewrite;$
$apply\ extensionality;$
$prove;$
$apply\ dlInBigU;$
$prove;$
$cases;$
$instantiate\ ss\_\_1\ ==\ locs\_of\ p\_\_0;$
$rewrite;$
$instantiate\ p\_\_1\ ==\ p\_\_0;$
$rewrite;$
$next;$
$split\ y\ \in\ locs\_of\ p;$
$rewrite;$
$cases;$
$instantiate\ ss\ ==\ locs\_of\ p;$
$rewrite;$
$instantiate\ p\_\_0\ ==\ p;$
$rewrite;$
$next;$
$instantiate\ ss\_\_1\ ==\ locs\_of\ p\_\_1;$
$rewrite;$
$instantiate\ p\_\_1\ ==\ p\_\_0;$
$rewrite;$
$next;$
$\blacksquare$

proof[$dlInLocs$]

$split\ x\ \in\ locs\,f;$
$cases;$
$apply\ dlLocsDef;$
$prove;$
$apply\ dlInBigU;$
$prove;$
$instantiate\ q\ ==\ p;$
$prove;$
$next;$
$rearrange;$
$split\ (\exists\ q:\ Piece\ \bullet\ q\ \in\ f\ \land\ x\ \in\ locs\_of\ q);$
$rewrite;$
$apply\ dlLocsDef;$
$prove;$
$apply\ dlInBigU;$
$prove;$
$instantiate\ ss\ ==\ locs\_of\ q;$
$prove;$
$instantiate\ p\ ==\ q;$
$prove;$
$next;$
$\blacksquare$

proof[$dlLocsCapEmpty$]

$split\ \ locs\_of\,(a)\ \cap\ locs\,(f)\ =\ \{\};$
$rewrite;$
$cases;$
$apply\ \ extensionality;$
$prove;$
$instantiate\ x\_\_0\ \ ==\ \ x;$
$rewrite;$
$apply\ \ dlInLocs;$
$rewrite;$
$instantiate\ q\ \ ==\ \ t;$
$rewrite;$
$next;$
$rearrange;$
$simplify;$
$split\ (\forall\ t:\ Piece\ \mid\ t\ \in\ f\ \bullet\ locs\_of\ \ a\ \cap\ locs\_of\ \ t\ =\ \{\});$
$rewrite;$
$apply\ \ extensionality;$
$prove;$
$apply\ \ dlInLocs;$
$prove;$
$instantiate\ t\ \ ==\ \ q;$
$rewrite;$
$instantiate\ x\_\_0\ \ ==\ \ x;$
$rewrite;$
$next;$
$\blacksquare$

proof[*dlNEW1Equiv*]
split NEW1;
rewrite;
cases;
prove by reduce;
split p.SIZE = s?;
cases;
rearrange;
instantiate t == p;
prove;
next;
prove;
instantiate t__0 == p;
prove;
next;
split NEW1Leo;
rewrite;
invoke NEW1Leo;
split NEW1Exact;
cases;
prove by reduce;
instantiate q == t;
instantiate p == t;
prove;
next;
simplify;
with disabled (NEW1Exact) prove by reduce;
instantiate q == t;
instantiate p == t;
prove;
next;
■

proof[*dlDISPOSE1Equiv*]
split DISPOSE1;
rewrite;
cases;
invoke;
prenex;
simplify;
instantiate join2 == join, np2 == np, abt2 == abt;
with enabled (DISPOSE1Join) invoke;
simplify;
next;
split DISPOSE1Leo;
rewrite;
invoke;
prenex;
trivial simplify;
with enabled (DISPOSE1Join) invoke;
instantiate join == join2, np == np2, abt == abt2;
rearrange;
simplify;
next;
■

proof[$dlDISPOSE1JoinWitness$]
$instantiate\ join2\ ==\ \{p?\} \cup \{\ r:\ Piece\ \mid\ r \in f1 \land p?\ \mathbf{abutt}\ r\ \}$,
$abt2\ ==\ \{\ r:\ Piece\ \mid\ r \in f1 \land p?\ \mathbf{abutt}\ r\ \}$,
$np2\ ==\ \theta\ Piece[LOC\ :=\ minLoc\,(\{p?\} \cup \{\ r:\ Piece\ \mid\ r \in f1 \land p?\ \mathbf{abutt}\ r\ \})$,
$SIZE\ :=\ sumSize\,(\{p?\} \cup \{\ r:\ Piece\ \mid\ r \in f1 \land p?\ \mathbf{abutt}\ r\ \})]$;
prove;
invoke $DISPOSE1Join$;
prove;
invoke *Piece*;
prove;
■

proof[$fDISPOSEJoinWithinHeap$]
invoke $DISPOSE1Join$;
prove ;
■

proof[$fDISPOSEJoinAbtWithinHeap$]
invoke $DISPOSE1Join$;
prove ;
■

proof[$lInAllLocs$]
apply *dlAllLocs*;
prove;
instantiate $p\ ==\ t$;
rewrite;
■

proof[$lAllLocsWithin$]
apply *inPower* *to predicate* $allLocs\,f \in \mathbb{P}\,(allLocs\,g)$;
prove ;
apply *dlAllLocs*;
prove;
instantiate $p\_\_0\ ==\ p$;
prove;
apply *inPower* *to predicate* $f \in \mathbb{P}\,g$;
instantiate $e\_\_0\ ==\ p$;
prove;
■

proof[$lAllLocResIsNotEmpty$]
apply *dlAllLocs*;
prove;
apply *extensionality*;
prove;
instantiate $x\_\_0\ ==\ x.LOC$;
prove;
instantiate $p\ ==\ x$;
prove;
■

proof[*lAllLocsDiff*]
apply extensionality;
prove;
cases;
apply dlAllLocs;
prove;
apply inPower to predicate $g \in \mathbb{P} f$;
instantiate $e == p\_\_0$;
*apply dlInFree*1 *to predicate* $f \in Free1$;
*apply dlInInvFree*1*Unique*;
instantiate $p1 == p,\ p2 == p\_\_0$;
prove;
next;
apply dlAllLocs;
prove;
instantiate $p\_\_1 == p$;
prove;
instantiate $p\_\_0 == p$;
prove;
next;
■

proof[*lMinLocsWithnAllLocs*]
apply dlMinLoc;
rewrite;
■

proof[*dlInFuse*]
with enabled (_ **fuse** _) *prove by reduce*;
■

proof[*dlFuseExcludedMiddle*]
with enabled (*dlInFuse*) *prove*;
■

proof[*dlInAbutt*]
with enabled (_ **abutt** _, *dlInFuse*) *prove by reduce*;
■

proof[*dlDISPOSE*1*JoinNoAbutt*]
apply extensionality;
prove;
*invoke DISPOSE*1*Join*;
rearrange;
*equality substitute join*2;
rewrite;
with normalization rewrite;
*equality substitute abt*2;
rewrite;
instantiate $w == x$;
rewrite;
■

proof[*dlDISPOSE*1*JoinAbuttBefore*]
apply extensionality;
prove;
*invoke DISPOSE*1*Join*;
rearrange;
*equality substitute join*2;
rewrite;
*equality substitute abt*2;
rewrite;
cases;
with normalization rewrite;
apply dlInAbutt to predicate ($p2$ **abutt** x);
rewrite;
rearrange;
with normalization rewrite;
instantiate $t == x$;
rewrite;
next;
with normalization rewrite;
apply dlInAbutt to predicate ($p2$ **abutt** y);
rewrite;
next;
∎

proof[*dlDISPOSE*1*JoinAbuttAfter*]
apply extensionality;
prove;
*invoke DISPOSE*1*Join*;
rearrange;
*equality substitute join*2;
rewrite;
*equality substitute abt*2;
rewrite;
cases;
with normalization rewrite;
apply dlInAbutt to predicate ($p2$ **abutt** x);
rewrite;
rearrange;
with normalization rewrite;
instantiate $t == x$;
rewrite;
next;
with normalization rewrite;
apply dlInAbutt to predicate ($p2$ **abutt** y);
rewrite;
next;
∎

proof[*lNoSelfAbutt*]
with enabled (*dlInAbutt*, *dlInFuse*) *prove by reduce*;
∎

proof[*lAllLocsUnit*]
apply extensionality;
apply dlAllLocs;
prove;
■

proof[*lMinLocUnit*]
apply dlMinLoc;
prove;
■

proof[*lSumSizeUnit*]
use dlSumSizeInduct$[ps\ :=\ \{\}]$;
with enabled (*dlSumSizeBase*) *prove*;
■

5.6 Heap0 to Heap1 refinement Proofs

proof[*Heap*1_ *vc*_ *ref*_ *fs*_ *init*]
instantiate $f1\_\_0' == f1', f0' == locs\, f1'$;
*invoke RetrFree*0*Free*1;
*invoke Init*0;
*invoke Heap*0;
*invoke Free*0;
invoke Loc;
rewrite;
apply extensionality;
prenex;
apply dlInLocs;
with disabled (*inRange*) *prove*;
conjunctive;
simplify;
cases;
apply dlLocsOfProp;
*invoke Init*1;
prove;
next;
prove by reduce;
apply dlLocsOfProp;
prove by reduce;
next;
■

proof[*NEW*1_ *vc*_ *ref*_ *fs*_ *applic*]
*use NEW*1_*vc*_*fsb*_*pre*;
rearrange;
rewrite;
■

proof[$DISPOSE1\_\ vc\_\ ref\_\ fs\_\ applic$]
  use $DISPOSE1\_vc\_fsb\_pre$;
  rearrange;
  rewrite;
  ■

5.7 Unfinished proofs

proof[*NEW1_ vc_ ref_ fs_ correct*]
prenex;
invoke RetrFree0Free1;
rewrite;
cases;
invoke Heap0;
invoke Free0;
invoke Loc;
rewrite;
next;
invoke NEW0;
invoke Δ *Heap0*;
invoke Heap0;
rewrite;
prenex;
invoke Free0;
invoke Loc;
rewrite;
equality substitute $f0$;
cases;
prove by reduce;
next;
apply dlNEW1Equiv;
invoke NEW1Leo;
split NEW1Exact;
simplify;
cases;
invoke NEW1Exact;
prenex;
equality substitute $f1'$;
apply extensionality to predicate $locs\ (f1 \setminus \{t\}) = locs\, f1 \setminus locs\_of\ r!$;
prenex;
rewrite;
cases;
apply dlLocsOfProp;
rewrite;
next;
apply dlLocsOfProp;
rewrite;
next;
invoke NEW1Bigger;
prenex;
equality substitute $f1'$;
rewrite;
apply extensionality to predicate $locs\ (\{rem\} \cup (f1 \setminus \{t\})) = locs\, f1 \setminus locs\_of\ r!$;
prenex;
rewrite;
cases;
cases;
apply dlInLocs;
rewrite;
prenex;
instantiate $q\_\_0 == q$;
rewrite;
rearrange;
rewrite;
instantiate $q\_\_0 == t$;
rearrange;
rewrite;
apply dlLocsOfProp to expression $locs\_of\ q$;
apply dlLocsOfProp to expression $locs\_of\ t$;
equality substitute q;

proof[$DISPOSE1\_vc\_ref\_fs\_correct$]

prenex;
invoke RetrFree0Free1;
rewrite;
cases;
invoke Heap0;
invoke Free0;
invoke Loc;
rewrite;
next;
invoke DISPOSE0;
invoke Δ *Heap0*;
invoke Heap0;
rewrite;
invoke Free0;
invoke Loc;
rewrite;
equality substitute $f0$;
apply dlDISPOSE1Equiv;
invoke DISPOSE1Leo;
prenex;
equality substitute $f1'$;
rewrite;
apply extensionality to predicate $locs\ (\{np2\} \cup (f1 \setminus join2)) = locs\_of\ p? \cup locs\, f1$;
prenex;
rewrite;
cases;
apply dlInLocs;
rewrite;
prenex;
instantiate $q\_\_0 == q$;
rewrite;
rearrange;
rewrite;
apply dlLocsOfProp to expression $locs\_of\ p?$;
apply dlLocsOfProp to expression $locs\_of\ q$;
rearrange;
equality substitute q;
rewrite;
rearrange;
  with normalization rewrite;
  cases;
use flDISPOSE1JoinCorrectnessNPBoundedSizeLemma;
rearrange;
equality substitute $p?$;
simplify;
use flDISPOSE1JoinCorrectnessNPMinLocLemma;
rewrite;
use dDISPOSE1CorrectnessLemmaFree1toFree0;
rearrange;
simplify;
next;
split $y \in locs\_of\ p?$;
simplify;
cases;
apply dlInLocs;
rewrite;
instantiate $q == p?$;
rearrange;
rewrite;
instantiate $q == np2$;
rewrite;
use dDISPOSE1CorrectnessLemmaFree0toFree1[$x := y$];

```
proof[lDISPOSE1FSBNonJoinPieceAfterNewPiece]
  split join2 = {p2}; cases;
  invoke DISPOSE1Join;
  rearrange;
  equality substitute join2;
  rearrange;
  split (p2 abutt r);
  cases;
  rewrite;
  apply extensionality to predicate {p2} = abt2 ∪ {p2 };
  equality substitute abt2;
  rewrite;
  instantiate y == r;
  prove;
  next;
  rearrange;
  rewrite;
  equality substitute np2;
  apply dlPieceBefore;
  apply dlLocsCapEmpty;
  rewrite;
  apply dlInAbutt to predicate p2 abutt r;
  apply dlInFuse;
  rewrite;
  instantiate t == r;
  apply extensionality to predicate locs_of p2 ∩ locs_of r = {};
  rewrite;
  instantiate x == r.LOC;
  rewrite;
  apply dlLocsOfProp to expression locs_of p2;
  apply dlLocsOfProp to expression locs_of r;
  rearrange;
  rewrite;
  next;

  use dlDISPOSE1JoinAbuttUnique;
  rearrange;
  rewrite;

  use dlDISPOSE1AbtValue;
  apply extensionality to predicate abt2 = {};
  prenex;
  rearrange;
  rewrite;
  equality substitute abt2;
  rewrite;
  apply dlInAbutt;
  rewrite;
  split x fuse p2;
  simplify;
  cases;
  use dlDISPOSE1JoinAbuttBeforeUnique[labt := x, other := r];
  rearrange;
  rewrite;
  apply dlPieceBefore;
  rearrange;
  rewrite;
  prenex;
  use dlDISPOSE1JoinValue;
  use dlDISPOSE1NP2LocValue;
  rearrange;
  rewrite;
  rearrange;
```

proof[*dlDISPOSE1JoinAbuttNotEmpty*]
$split\ \forall\ k : Piece \mid k \in fr \bullet \lnot\ p2$ **abutt** k;
simplify;
cases;
apply extensionality to predicate $abt2 = \{\}$;
prove;
invoke $DISPOSE1Join$;
equality substitute $abt2$;
rewrite;
next;
simplify;
apply extensionality to predicate $abt2 = \{\}$;
prove;
next;
split $DISPOSE1Join$;
simplify;
rearrange;
simplify;
prove;
■

proof[*lDISPOSE1FSBAuxLemma*]
split $p2 \in fr$;
cases;
apply dlLocsCapEmpty ;
rewrite;
instantiate $t == p2$;
rewrite;
apply capSubsetRight ;
rewrite;
apply extensionality;
apply dlLocsOfProp ;
prove;
instantiate $x == p2.LOC$;
prove;
next;
apply cupCommutes;
with disabled (*cupCommutes*) *rewrite*;
next;
■

5.8 Commented proofs

This section contains partial proof attempts at various goals. It was the beginnings of PP data collection andi is here for reference/history?

This was the proof when $SIZE \geq 0$

```
begin{zproof}[NEW1\_vc\_fsb\_pre]
prove by reduce;
%%
% No longer needed, given we have the result explicitly assigned to this witness now
%%
% cook up witnesses for r!, as we will use q for p, use q.LOC
%split \lnot \exists result: Piece @
% result = \theta Piece[LOC := q.LOC, SIZE := s?] ;
%cases;
% with enabled (Loc, Piece\$member) prove by reduce;
%next;
%prenex;
```

```
%rearrange;
% cook up a witness for rem which take into account result
% (i.e. return the remainder of result, if any)? So, we need
% two cases, one where the remainder is empty, and another
% where there is a remainder.
split q.SIZE = s?;
rewrite;
% case1: when there is no remainder, q is result and remainder is empty
cases;
rearrange;
%%
% No longer needed, given we already have the remainder cupped with f and p as the witness below
%%
%% cook up a witness for an empty remainder starting at some location
%split \lnot \exists someLoc: Loc; remainder : Piece @ remainder = \theta Piece[LOC := someLoc, SIZE := 0];
%cases;
% with enabled (Loc, Piece\$member) prove by reduce;
% instantiate someLoc == 0;
% prove;
%next;
%prenex;
%rearrange;
%instantiate res! == result, p == q, rem == remainder;
% KEY POINT1: now we have the empty remainder and the result witness
instantiate p == q ;
reduce;
apply lFree1UnitUnionInType;
reduce;
%%
% No need given there is no subset on locsof/locs
%%
%apply dlLocsOfWithin;
%apply dlLocsEmptyRemainder to expression \locsOf remainder;
%rearrange;
%rewrite;
% we are done: q is result, hence is empty;
%split \lnot q = result;
%cases;
% with enabled (Piece\$member) prove by reduce;
%next;
% equality substitute result;
% apply diffSuperset;
% rewrite;
    next;
     prove;
% case2: when there is a remainder, q is result and remainder is the right size
%%
% No longer needed, given we already have the remainder cupped with f and p as the witness below
%%
% cook up a witness for an empty remainder starting at some location
%split \lnot \exists remainder : Piece @ remainder = \theta Piece[LOC := q.LOC + req?, SIZE := q.SIZE - req?];
%cases;
% with enabled (Loc, Piece\$member) prove by reduce;
%next;
%prenex;
%rearrange;
%instantiate res! == result, p == q, rem == remainder;
% KEY POINT2: now we have the empty remainder and the result witness
instantiate p == q;
reduce;
%apply dlLocsOfWithin;
%rewrite;
%use dlLocsRemainder[rem := remainder, q := q, res := result];
%rearrange;
%rewrite;
next;
end{zproof}

begin{zproof}[DISPOSE1\_vc\_fsb\_pre]
apply dlDISPOSE1Equiv;
prove by reduce;
% Instead of the complicated witnessing process below, I just use a lemma
use dlDISPOSE1JoinWitness;
prove by reduce;
instantiate join2\_\_0 == join2, np2\_\_0 == np2;
```

```
prove;
% NOTE: interesting note on shape
% that's annoying: because the witness for the cup in NEW1\_pre is an explicit record
% the expression "f \setminues \{q\} \cup \{RECORD\}" keeps the RECORD on the RHS, hence
% the shape for the lFree1UnitUnionInType as "f \cup \{t\} \in Free1".
% Here, for DISPOSE1, because the witness is an element np, the
% "f \setminus \{q\} \cup \{np\}" rewrites to "\{np\} \cup (f \setminus \{q\})"
% We can't have both copies of the lemma, though to avoid looping. Okay fine.
apply cupCommutes;
with disabled (cupCommutes) rewrite;
apply lFree1UnitUnionInType ;
apply dlLocsCapEmpty ;
prove;
     % try something crazy on the only added invariant about DISPOSE1; save for latter
     %apply dlLocsOfCapEmpty ;
     %instantiate t == p;
     %rearrange;
     %rewrite;
conjunctive;
rewrite;
cases;
% NOTE: STRATEGY REPEATED from NEW1\_pre (mostly in intent, not in structure)!!!
     % chosen witness doesn't share locations with others in f
     %
use fDISPOSEJoinWithinHeap;
     invoke DISPOSE1Join;
     equality substitute np2;
     rearrange;
     rewrite;
     apply lMinLocsWithnAllLocs to predicate minLoc~join2 \in allLocs (f1 \setminus join2);
     rewrite;
    next;
     USE An AUX LEMMA INSTEAD
end{zproof}

begin{zproof}[DISPOSE1\_vc\_fsb\_pre]
apply dlDISPOSE1Equiv;
prove by reduce;
% Instead of the complicated witnessing process below, I just use a lemma
use dlDISPOSE1JoinWitness;
prove by reduce;
instantiate join2\_\_0 == join2, np2\_\_0 == np2;
prove;
% NOTE: interesting note on shape
% that's annoying: because the witness for the cup in NEW1\_pre is an explicit record
% the expression "f \setminues \{q\} \cup \{RECORD\}" keeps the RECORD on the RHS, hence
% the shape for the lFree1UnitUnionInType as "f \cup \{t\} \in Free1".
% Here, for DISPOSE1, because the witness is an element np, the
% "f \setminus \{q\} \cup \{np\}" rewrites to "\{np\} \cup (f \setminus \{q\})"
% We can't have both copies of the lemma, though to avoid looping. Okay fine.
apply cupCommutes;
with disabled (cupCommutes) rewrite;
apply lFree1UnitUnionInType ;
apply dlLocsCapEmpty ;
prove;
     % try something crazy on the only added invariant about DISPOSE1; save for latter
     %apply dlLocsOfCapEmpty ;
     %instantiate t == p;
     %rearrange;
     %rewrite;
conjunctive;
rewrite;
cases;
% NOTE: STRATEGY REPEATED from NEW1\_pre (mostly in intent, not in structure)!!!
     % chosen witness doesn't share locations with others in f
     %
apply lAllLocsDiff to expression allLocs (f1 \setminus join2);
rewrite;
     use fDISPOSEJoinWithinHeap;
     invoke DISPOSE1Join;
     equality substitute np2;
     rearrange;
     rewrite;
     % now bring the lemmas about min S within S when s is not empty
     apply lMinLocsWithnAllLocs to predicate minLoc~join2 \in allLocs~join2;
```

```
     rewrite;
     % join2 isn't empty, so the case for join2 \in \power fr is proved.
     % just neeed the second case.
     split join2 \in \power f1;
     cases;
     prove;
     next;
     rewrite;
     apply inPower to predicate join2 \in \power~f1;
     apply inPower to predicate join2 \setminus \{p2\} \in \power~fr;
     prenex;
     instantiate e\_\_0 == e;
     rearrange;
     rewrite;
     prove;
     rearrange;
     prenex;
     % Go back and add the lemma from the previous proof that join2 is a subset of f1, hence
     % if p is in f1 and not in join2?
     rearrange;
     % to avoid too much eq substitution rearrange terms more aggressively/explicitly
end{zproof}

begin{zproof}[DISPOSE1\_vc\_fsb\_pre]
apply dlDISPOSE1Equiv;
prove by reduce;
% Instead of the complicated witnessing process below, I just use a lemma
use dlDISPOSE1JoinWitness;
prove by reduce;
instantiate join2\_\_0 == join2, np2\_\_0 == np2;
prove;
% NOTE: interesting note on shape
% that's annoying: because the witness for the cup in NEW1\_pre is an explicit record
% the expression "f \setminues \{q\} \cup \{RECORD\}" keeps the RECORD on the RHS, hence
% the shape for the lFree1UnitUnionInType as "f \cup \{t\} \in Free1".
% Here, for DISPOSE1, because the witness is an element np, the
% "f \setminus \{q\} \cup \{np\}" rewrites to "\{np\} \cup (f \setminus \{q\})"
% We can't have both copies of the lemma, though to avoid looping. Okay fine.
apply cupCommutes;
with disabled (cupCommutes) rewrite;
apply lFree1UnitUnionInType ;
apply dlLocsCapEmpty ;
prove;
     % try something crazy on the only added invariant about DISPOSE1; save for latter
     %apply dlLocsOfCapEmpty ;
     %instantiate t == p;
     %rearrange;
     %rewrite;
conjunctive;
rewrite;
cases;
% NOTE: STRATEGY REPEATED from NEW1\_pre (mostly in intent, not in structure)!!!
     % chosen witness doesn't share locations with others in f
     %
     invoke DISPOSE1Join;
     equality substitute np2;
     rewrite;
     apply dlAllLocs;
     rewrite;
     prenex;
     % Go back and add the lemma from the previous proof that join2 is a subset of f1, hence
     % if p is in f1 and not in join2?
     rearrange;
     % to avoid too much eq substitution rearrange terms more aggressively/explicitly
     split f1 \in Free1 \land p? \in Piece \land fr = f1 \land p2 = p? \land
     join2 \in \power Piece \land
     np2 = \theta (Piece [LOC := minLoc~join2, SIZE := sumSize~join2]) \land
     np2 \in Piece \land p \in Piece \land p \in f1 \land
     (\forall t: Piece | t \in f1 @ \locsOf~p? \cap \locsOf~t = \{\}) \land
     \lnot f1 \setminus join2 \cup \{\theta (Piece [LOC := minLoc~join2, SIZE := sumSize~join2]) \} \in Free1 \land
      minLoc~join2 = p . LOC \land
     join2 = \{p2 \} \cup \{q: Piece | q \in fr \land p2 \abutt q \} \land
     \lnot p \in join2 ;
     simplify;
```

```
     % Bring in the properties about DISPOSEJoin1 to avoid needing to expand it
     use fDISPOSEJoinWithinHeap;
     rearrange;
     apply inPower;
     with predicate (DISPOSE1Join) simplify;
     apply lInAllLocs;
     rewrite;
     %lMinLocsWithnAllLocs
     use fDISPOSEJoinMinWithinJoin;
     use fDISPOSEJoinLocsWithinHeapLocs;
     simplify;
     rewrite;
     apply dlAllLocs;
     rewrite;
     prenex;
end{zproof}
begin{zproof}[DISPOSE1\_vc\_fsb\_pre]
apply dlDISPOSE1Equiv;
prove by reduce;
% Instead of the complicated witnessing process below, I just use a lemma
use dlDISPOSE1JoinWitness;
prove by reduce;
instantiate join2\_\_0 == join2, np2\_\_0 == np2;
prove;
% NOTE: interesting note on shape
% that's annoying: because the witness for the cup in NEW1\_pre is an explicit record
% the expression "f \setminues \{q\} \cup \{RECORD\}" keeps the RECORD on the RHS, hence
% the shape for the lFree1UnitUnionInType as "f \cup \{t\} \in Free1".
% Here, for DISPOSE1, because the witness is an element np, the
% "f \setminus \{q\} \cup \{np\}" rewrites to "\{np\} \cup (f \setminus \{q\})"
% We can't have both copies of the lemma, though to avoid looping. Okay fine.
apply cupCommutes;
with disabled (cupCommutes) rewrite;
apply lFree1UnitUnionInType ;
apply dlLocsCapEmpty ;
prove;
     % try something crazy on the only added invariant about DISPOSE1; save for latter
     %apply dlLocsOfCapEmpty ;
     %instantiate t == p;
     %rearrange;
     %rewrite;
conjunctive;
rewrite;
cases;
% NOTE: STRATEGY REPEATED from NEW1\_pre (mostly in intent, not in structure)!!!
     % chosen witness doesn't share locations with others in f
     apply dlAllLocs;
     rewrite;
     prenex;
     invoke DISPOSE1Join;
     equality substitute np2;
     rewrite;
     % Go back and add the lemma from the previous proof that join2 is a subset of f1, hence
     % if p is in f1 and not in join2?
     rearrange;
     % to avoid too much eq substitution rearrange terms more aggressively/explicitly
     split f1 \in Free1 \land p? \in Piece \land fr = f1 \land p2 = p? \land
     join2 \in \power Piece \land
     np2 = \theta (Piece [LOC := minLoc~join2, SIZE := sumSize~join2]) \land
     np2 \in Piece \land p \in Piece \land p \in f1 \land
     \locsOf p? \cap locs~f1 = \{\} \land
     \lnot f1 \setminus join2 \cup \{\theta (Piece [LOC := minLoc~join2, SIZE := sumSize~join2]) \} \in Free1 \land
      minLoc~join2 = p . LOC \land
     join2 = \{p2 \} \cup \{q: Piece | q \in fr \land p2 \abutt q \} \land
     \lnot p \in join2 ;
     simplify;
     equality substitute join2;
     rewrite;
     apply dlMinLocInduct to expression minLoc~join2;
     apply dlAllLocs to expression allLocs~join2;
     apply dlInFree1 to predicate f1 \in Free1;
     apply dlInInvFree1Sep;
prenex;
     instantiate p1 == q, p2 == p;
     prove;
```

```
     apply dlPieceBefore;
     rewrite;
    next;
     % chosen witness is before other elements of f
     apply dlInFree1 to predicate f1 \in Free1;
     apply dlInInvFree1Sep;
     apply dlPieceBefore;
     reduce;
     instantiate p1 == q, p2 == r;
     prove;
    next;
     % chosen witness is after other elements of f
     apply dlInFree1 to predicate f1 \in Free1;
     apply dlInInvFree1Sep;
     instantiate p1 == r, p2 == q;
     rearrange;
     rewrite;
     split r . LOC < q . LOC;
     rewrite;
     cases;
     % when the remainder is whole before q (no overlapping)
     apply dlPieceBefore;
     reduce;
     next;
     rearrange;
     split r.LOC = q.LOC;
     cases;
     % there is no overlap, so r=q, because of uniqueness over f
     apply dlInInvFree1Unique;
     rearrange;
     instantiate p1 == r, p2 == q;
     prove;
     next;
     % there can't be any other case: if r is not q, and is before q+s?
     % and also after q, then there is an inconstency of the invariant,
     % hence the contradiction (i.e. all cases have already been covered)
     rearrange;
     simplify;
     instantiate p1 == q, p2 == r;
     prove;
     apply dlPieceBefore;
     rewrite;
next;
end{zproof}

begin{zproof}[DISPOSE1\_vc\_fsb\_pre]
apply dlDISPOSE1Equiv;
with disabled (cupCommutes) prove by reduce;
% NOTE: the Z construct cannot be translated error! Annoying
%
% need argument for the finiteness of j
split \lnot \{ q: Piece | q \in f1 \land (p?.LOC+p?.SIZE = q.LOC \lor q.LOC+q.SIZE = p?.LOC) \} \subseteq f1;
%split \lnot \forall q: Piece | q \in f1 \land q \abutt p? @ p? \in f1;
%split \lnot \{ q: Piece | q \in f1 \land q \abutt p? \} \subseteq f1;
cases;
prove;
next;
rearrange;
% pick the witnesses from within DISPOSE1Join
split \lnot \exists j : \power~Piece @
j = \{ p? \} \cup \{ q: Piece | q \in f1 \land (p?.LOC+p?.SIZE = q.LOC \lor q.LOC+q.SIZE = p?.LOC) \};
%\land
% this is needed to expose the fact j is finite; declaring as such makes it more difficult to prove
%j \subseteq f \setminus \{ p?\};
cases;
prove;
next;
% avoi doing prove to avoid loosing information that j \in \power~Piece
prenex;
rearrange;
% now get that j is finite from the fact that the other set is finite
split \lnot j \in \finset~Piece;
cases;
prove;
next;
```

```
rearrange;
simplify;
% now we know that j is finite and it has the right value for DISPOSE1Join
instantiate join2 == j, np2 == \theta Piece[LOC := minLoc~(j), SIZE := sumSize~(j)] ;
invoke DISPOSE1Join;
reduce;
% needs a lemma about finite set subseting
next;
end{zproof}

begin{zproof}[lDISPOSE1FSBNonJoinPieceAfterNewPiece]
use dlDISPOSE1JoinNoAbuttV3;
rearrange;
simplify;
invoke DISPOSE1Join;
rearrange;
% first case is easy: p2 cannot abutt r
split (p2 \abutt r);
cases;
prove;
next;
% real case: non-abutting r means join is just p2
simplify;
rearrange;
% NOTE-1: arranging terms explicitly to avoid eq-subst / rewrite over terms
%   (i.e. influencing the proof context; or something like a tactical feature?)
split join2 = \{p2\} \land np2 = \theta (Piece [LOC := minLoc~join2, SIZE := sumSize~join2]);
rewrite;
equality substitute join2;
equality substitute np2;
apply dlPieceBefore;
rewrite;
apply dlMinLoc;
% NOTE-1: first lemma about sumSize for singletons or in general actually
%   \forall ps: \finset_1~Piece @ sumSize~ps = ...
%apply dlSumSize;
use dlSumSizeInduct[p:=p2, ps := \{\}];
rearrange;
with enabled (dlSumSizeBase) rewrite;
% NOTE-1: suggested the lemma about allLocs unit
apply dlInAbutt to predicate (p2 \abutt r);
apply dlLocsOfCapEmpty to expression \locsOf p2 \cap \locsOf r;
rewrite;
apply dlPieceExcludedMiddle ;
rewrite;
end{zproof}
begin{zproof}[lDISPOSE1FSBNonJoinPieceAfterNewPiece]
rewrite;
% NOTE-1: the non-abuttness lemma as I created it not being used. It doesn't quite have
%   the shape I need, though.
use dlDISPOSE1JoinNoAbutt;
rearrange;
simplify;
split join2 = \{p2\};
rewrite;
cases;
% if join is just p, then abt is empty;
split \lnot abt2 = \{\};
cases;
invoke DISPOSE1Join;
rearrange;
equality substitute join2;
apply extensionality to predicate abt2 = \{\};
apply extensionality to predicate  \{p2\} =   \{p2\} \cup abt2;
prenex;
rewrite;
instantiate y == x;
prove;
% NOTE-1: suggested the lemma lNoSelfAbutt, that  kicks in here.
next;
% this is where I started of with, but now I have the invariant about abt2 = empty
invoke DISPOSE1Join;
equality substitute join2;
equality substitute abt2;
rewrite;
```

```
apply extensionality to predicate \{\} = \{q: Piece | q \in fr \land p2 \abutt q \};
rewrite;
instantiate y == r;
rearrange;
rewrite;
% NOTE-1: This suggests another format (new side conditions!!!) for the abuttingness lemmas
%   abt2 = \{\}, abt2 = \{ labt \}, abt2 = \{ rabt \} etc...
next;
next;
end{zproof}

\begin{zproofold}[lDISPOSE1FSBAuxLemma]
apply dlLocsCapEmpty ;
rewrite;
% can't be true
split p2 \in fr;
cases;
instantiate t == p2;
rewrite;
apply capSubsetRight ;
rewrite;
apply extensionality;
apply dlLocsOfProp ;
prove;
instantiate x == p2.LOC;
prove;
% TODO: maybe add a lemma that says \forall p: Piece @ \lnot \locsOf~p = \{\}?
next;
apply cupCommutes;
with disabled (cupCommutes) rewrite;
apply lFree1UnitUnionInType;
conjunctive;
rewrite;
cases;
% NOTE: inverted the equality in the lemma to approrpialy match the goal... didn't work
% this is because now I realised that the locations of abt should be the same as join
%
use dlDISPOSE1AbutLocJoins;
rearrange;
simplify;
split allLocs~abt2 = allLocs~join2;
simplify;
%NOTE: with the addition of fDISPOSEJoinAbtWithinHeap and the
%    change to have abt instead of join in DISPOSE1, the
%     lemma lAllLocsDiff now kicks in automatically.
%apply lAllLocsDiff to expression allLocs (fr \setminus abt2);
invoke DISPOSE1Join;
equality substitute np2;
equality substitute allLocs~abt2;
rewrite;
% WHY DOESNT THIS APPLY AUTOMATICALLY?
apply lMinLocsWithnAllLocs;
rewrite;
prove;
next;
% handling new piece before/after
prove;
with normalization rewrite;
cases;
use dlDISPOSE1JoinNoAbutt;
rearrange;
simplify;
invoke DISPOSE1Join;
rearrange;
equality substitute join2;
equality substitute abt2;
rewrite;
instantiate t == r;
apply dlInAbutt to predicate (p2 \abutt r);
apply dlInFuse;
rewrite;
apply dlLocsOfCapEmpty to expression \locsOf p2 \cap \locsOf r;
rewrite;
apply dlPieceExcludedMiddle ;
rewrite;
```

```
with normalization rewrite;
% case when np is before other non-abutting pieces r in f
cases;
% case when np is before, yet r is before as well = contradiction
cases;
apply dlPieceBefore;
rewrite;
% case when np.LOC < r.LOC yet not np before r and not r before np! Contradiction
next;
apply dlPieceBefore;
rewrite;
apply lLessNeg to predicate np2 . LOC + np2 . SIZE < r . LOC;
rewrite;
split np2.LOC + np2.SIZE = r.LOC;
rewrite;
cases;
next;
next;
cases;
apply dlPieceBefore;
rewrite;
next;
split np2.LOC = r.LOC;
rewrite;
apply dlPieceBefore;
rewrite;
next;
next;
next;
next;
next;
\end{zproofold}
\begin{zproofold}[lDISPOSE1FSBAuxLemma]
apply dlLocsCapEmpty ;
rewrite;
% can't be true
split p2 \in fr;
cases;
instantiate t == p2;
rewrite;
apply capSubsetRight ;
rewrite;
apply extensionality;
apply dlLocsOfProp ;
prove;
instantiate x == p2.LOC;
prove;
% TODO: maybe add a lemma that says \forall p: Piece @ \lnot \locsOf~p = \{\}?
next;
apply cupCommutes;
with disabled (cupCommutes) rewrite;
apply lFree1UnitUnionInType;
conjunctive;
rewrite;
cases;
% NOTE: inverted the equality in the lemma to approrpialy match the goal... didn't work
% this is because now I realised that the locations of abt should be the same as join
%
use dlDISPOSE1AbutLocJoins;
rearrange;
simplify;
split allLocs~abt2 = allLocs~join2;
simplify;
%NOTE: with the addition of fDISPOSEJoinAbtWithinHeap and the
%    change to have abt instead of join in DISPOSE1, the
%     lemma lAllLocsDiff now kicks in automatically.
%apply lAllLocsDiff to expression allLocs (fr \setminus abt2);
invoke DISPOSE1Join;
equality substitute np2;
equality substitute allLocs~abt2;
rewrite;
% WHY DOESNT THIS APPLY AUTOMATICALLY?
apply lMinLocsWithnAllLocs;
rewrite;
prove;
```

```
next;
prove;
apply dlPieceExcludedMiddle ;
rewrite;
with normalization rewrite;
% case when np is before other non-abutting pieces r in f
cases;
% case when np is before, yet r is before as well = contradiction
cases;
apply dlPieceBefore;
rewrite;
% case when np.LOC < r.LOC yet not np before r and not r before np! Contradiction
next;
next;
cases;
apply dlPieceBefore;
rewrite;
next;
split np2.LOC = r.LOC;
rewrite;
apply dlPieceBefore;
rewrite;
next;
next;
next;
next;
next;
\end{zproofold}
\begin{zproofold}[lDISPOSE1FSBAuxLemma]
apply dlLocsCapEmpty ;
rewrite;
% can't be true
split p2 \in fr;
cases;
instantiate t == p2;
rewrite;
apply capSubsetRight ;
rewrite;
apply extensionality;
apply dlLocsOfProp ;
prove;
instantiate x == p2.LOC;
prove;
% TODO: maybe add a lemma that says \forall p: Piece @ \lnot \locsOf~p = \{\}?
next;
apply cupCommutes;
with disabled (cupCommutes) rewrite;
apply lFree1UnitUnionInType;
conjunctive;
rewrite;
cases;
% the definition for join2 is unfortunate.. as it makes the proof harder
% due to the possible counter example when the elements removed are not within fr
split join2 \in \power fr;
cases;
rewrite;
invoke DISPOSE1Join;
equality substitute np2;
rewrite;
apply lMinLocsWithnAllLocs;
prove;
next;
use fDISPOSEJoinWithinHeap;
apply inPower;
rearrange;
prenex;
simplify;
instantiate e\_\_0 == e;
rewrite;
invoke DISPOSE1Join;
equality substitute np2;
rewrite;

apply lAllLocsDiff;
prove;
```

```
split join2 \in \power fr;
simplify;
apply inPower;
invoke DISPOSE1Join;
equality substitute np2;
     rearrange;
     rewrite;
     apply lMinLocsWithnAllLocs to predicate minLoc~join2 \in allLocs~join2;
     rewrite;
     apply lMinLocsWithnAllLocs to predicate minLoc~join2 \in allLocs (f1 \setminus join2);
     rewrite;
rearrange;
simplify;
rewrite;
prove;
apply dlInFree1;
rewrite;
\end{zproofold}


\begin{zproofold}[lFree1UnitUnionInType]
split f \cup \{p\} \in Free1;
rewrite;
cases;
apply dlInFree1;
with enabled (dlPieceWellPlaced) with normalization prove;
cases;
instantiate q == t;
rewrite;
next;
apply dlInInvFree1Sep to predicate \sep~(f \cup \{ p\});
apply dlPieceBefore;
rewrite;
instantiate p1 == t, p2 == p;
instantiate p1 == p, p2 == t;
prove;
% After adding types to the existential quantifier
%
%with normalization rewrite;
%% this true. just nee the info for t in Piece
%apply inPower;
%instantiate e == t;
%with normalization with enabled (Piece\$member) prove by reduce;
next;
simplify;
rearrange;
split (\forall t: Piece | t \in f @ p \wellplaced t);
rewrite;
apply dlInFree1;
apply dlInInvFree1Sep;
apply dlInInvFree1Unique;
rearrange;
rewrite;
% changing dlInInvFree1Sep to include the type for p1,p2 makes a world of difference! Oh my...
%cases;
% rearrange;
% instantiate p1\_\_0 == p1, p2\_\_0 == p2;
% prove;
% with normalization rewrite;
% cases;
% rearrange;
% instantiate t == p1;
% with enabled (dlPieceWellPlaced, dlPieceBefore) with normalization prove;
% next;
% cases;
% rearrange;
% instantiate t == p2;
% with enabled (dlPieceWellPlaced, dlPieceBefore) with normalization prove;
% cases;
% % because sep invariant doesn't include p1 \in fr etc....
% apply inPower;
% instantiate e == p2;
% with normalization with enabled (Piece\$member) prove by reduce;
% next;
% instantiate t == p2;
```

```
% rearrange;
% simplify;
% apply inPower;
% instantiate e == p2;
% prove;
% next;
% apply dlInInvFree1Unique;
% split p2 = p;
% apply dlPieceWellPlaced to predicate p \wellplaced p1;
% prove;
% cases;
% with normalization rewrite;
%next;
% instantiate t == q;
% with enabled (dlPieceWellPlaced) prove;
% with normalization rewrite;
% apply dlPieceBefore to predicate q \before p;
% rewrite;
\end{zproofold}
\begin{zproofold}[lFree1UnitUnionInType]
  prove;
  apply dlInFree1;
  prove;
  apply dlInInvFree1Sep ;
  with disabled (inCup) prove;
  instantiate p1\_\_0 == p1, p2\_\_0 == p2;
  with normalization rewrite;
  cases;
instantiate p1\_\_0 == p1;
rewrite;
instantiate p2\_\_0 == p2;
rewrite;
%     use ulLocsOfDisjFreeInvProp1;
%     use ulLocsOfDisjFreeInvProp2;
%     prove;
%     % this need here to expand Free1's inv in the middle of proof is annoying
%     apply dlInFree1;
%     apply dlInInvFree1Sep;
%     prove;
%     instantiate p1\_\_2 == p1\_\_0, p2\_\_2 == p2\_\_0;
%     instantiate p1\_\_2 == p1\_\_1, p2\_\_2 == p2\_\_1;
%     prove;
  next;
   cases;
%     use ulLocsOfDisjFreeInvProp1[p1 := p2, p2 := p1];
%     use ulLocsOfDisjFreeInvProp3[p1 := p2, p2 := p1];
%     prove;
      % this need here to expand Free1's inv in the middle of proof is annoying
%     apply dlInFree1;
%     apply dlInInvFree1Sep;
%     prove;
%     instantiate p1\_\_2 == p1\_\_0, p2\_\_2 == p2\_\_0;
%     instantiate p1\_\_2 == p1\_\_1, p2\_\_2 == p2\_\_1;
%     prove;
%     apply inPower;
%     instantiate e == p2;
%     with enabled (Piece\$member, Loc) prove by reduce;
  next;
\end{zproofold}

\begin{zproofold}[lLocsDistUnitDiff]
  apply dlLocsDef ;
  % I already had a proof with using my specialised bigU...
  with disabled (dlBigCupAsBigU) prove;
  apply extensionality;
  with disabled (dlBigCupAsBigU) prove;
  apply inBigcup;
  prove;
  cases;
  % ANNOYING!
  split \lnot x \in \num;
  cases;
  prove;
  next;
  rewrite;
```

```
  instantiate B\_\_0 == B;
  prove;
  instantiate p\_\_1 == p\_\_0;
  prove;
  %% INTERESTING: the proof of uniqueness, finally surfaces!
  %% THANKS AV: I need another lemma to show the disjointness of invFree and \locsOf
  apply dlInFree1;
  apply dlInInvFree1Sep;
  instantiate p1 == p, p2== p\_\_0;
  prove;
  apply dlLocsOfProp;
  prove;
  apply inPower;
  instantiate e == p;
  with enabled (Piece\$member, Loc) prove by reduce;
  next;
  split \lnot y \in \num;
  cases;
  prove;
  next;
  rewrite;
  instantiate B == B \_\_0;
  prove;
  instantiate p\_\_1 == p\_\_0;
  prove;
  next;
\end{zproofold}
\begin{zproofold}[lAllLocsDiff]
apply extensionality;
prove;
cases;
% because of lAllLocsWithin it gets the case for in allLocs f
apply dlAllLocs;
prove;
apply dlInFree1 to predicate f \in Free1;
apply dlInInvFree1Unique;
instantiate p1 == p, p2 == p\_\_0;
prove;
with normalization rewrite;
next;
apply dlAllLocs;
prove;
instantiate p\_\_1 == p;
prove;
instantiate p\_\_0 == p;
prove;
next;
\end{zproofold}
\begin{zproofold}[lAllLocsIsFinset1]
use applyInRanPfun[\power~\lblot LOC: \num; SIZE : \num \rblot, \power~\num][A := \power~Piece, B := \finset~\num, f := a
rearrange;
rewrite;
with normalization rewrite;
invoke (\_ \pfun \_);
rewrite;
conjunctive;
rewrite;
cases;
invoke (\_ \rel \_);
apply inPower;
prenex;
rewrite;
apply inCross2;
use allLocs\$declaration;
invoke (\_ \fun\_);
rewrite;
instantiate x == f;
prenex;
use pairInFunction[\power~Piece, \power~Loc][f := allLocs, x := f, y := y];
rearrange;
rewrite;
rearrange;
rewrite;
instantiate x\_\_0 == f;
rearrange;
```

```
rewrite;
instantiate y\_\_1 == y;
rewrite;
instantiate x\_\_0 == x;
rearrange;
rewrite;
prenex;
rewrite;
invoke (\_ \pfun\_);
rewrite;
invoke (\_ \rel \_);
split allLocs~f \in \finset \num;
rewrite;
apply inFinset1 ;
invoke (\finset~\_);
cases;
trivial rewrite;
rearrange;
prenex;
instantiate S\_\_0 == allLocs~f;
rewrite;
instantiate n\_\_0 == n;
rewrite;
instantiate n == \#~f;
rewrite;
use cardIsNonNegative[\lblot LOC: \num; SIZE: \num \rblot][S := f];
rearrange;
rewrite;
instantiate f\_\_0 == (\lambda x: 1 \upto n @ allLocs~f);
invoke (\finset~ \_);
trivial rewrite;
prenex;
rearrange;
instantiate n\_\_0 == n, f\_\_1 == f\_\_0;
prove;
next;
prove;
\end{zproofold}
\begin{zproofold}[NEW1\_ vc\_ ref\_ fs\_ correct]
prenex;
invoke RetrFree0Free1;
rewrite;
cases;
% state data refinement is correct
invoke Heap0;
invoke Free0;
invoke Loc;
rewrite;
next;
% operation refinement: just the easy / non-changing bits
invoke NEW0;
invoke \Delta Heap0;
invoke Heap0;
rewrite;
prenex;
invoke Free0;
invoke Loc;
rewrite;
% now linking th real functionality
equality substitute f0;
invoke NEW1;
prenex;
equality substitute f1';
cases;
% input's use to the result on both cases is refined
prove;
next;
% state update result is refined.
rewrite;
apply extensionality to predicate locs (\{rem \} \cup (f1 \setminus \{p\_\_0 \})) = locs~f1 \setminus \locsOf r!;
prenex;
rewrite;
% could apply dlInLocs everywhere, but go slowly first
cases;
% case1: NEW1.f1' subseteq NEW0.f0'
```

```
cases;
%  x \in locs (\{rem \} \cup (f1 \setminus \{p\_\_0 \})) \implies x \in locs~f1
apply dlInLocs;
rewrite;
prenex;
instantiate q\_\_1 == q\_\_0;
rewrite;
rearrange;
rewrite;
% residual part is to show that links NEW1.p and rem
instantiate q\_\_1 == p;
rearrange;
rewrite;
apply dlLocsOfProp to expression \locsOf~q\_\_0;
apply dlLocsOfProp to expression \locsOf~p;
equality substitute q\_\_0;
equality substitute rem;
rewrite;
next;
% not within the result of returned piece
%  x \in locs (\{rem \} \cup (f1 \setminus \{p\_\_0 \})) \implies \lnot x \in \locsOf~r!
apply dlInLocs;
rewrite;
prenex;
split q\_\_0 = rem;
rewrite;
cases;
apply dlLocsOfProp to expression \locsOf~q\_\_0;
apply dlLocsOfProp to expression \locsOf~r!;
rearrange;
equality substitute q\_\_0;
equality substitute rem;
equality substitute r!;
rewrite;
next;
% now the reminder is within f1, so that's a contradiction because so is NEW1.p
% use the Free1 invariant
invoke;
apply dlInFree1;
apply dlInInvFree1Sep;
instantiate p1 == p\_\_0, p2 == q\_\_0;
% ZEVes unhelpfully change the names above - they become more "imporant". it's the same instantiation below
instantiate p2 == p, p1 == q;
rearrange;
rewrite;
next;
% case2: NEW0.f0' subseteq NEW1.f1'
apply dlInLocs;
rewrite;
prenex;
instantiate q\_\_1 == q\_\_0;
rewrite;
rearrange;
% need the invariant for Free1 over q\_\_0 and p?
invoke;
apply dlInFree1;
apply dlInInvFree1Sep;
instantiate p1 == p, p2 == q\_\_0;
instantiate p2 == p, p1 == q\_\_0;
rearrange;
rewrite;

% DEAD END: missing link with Free1Inv
%apply dlLocsOfProp to expression \locsOf~r!;
%apply dlLocsOfProp to expression \locsOf~q\_\_0;
%apply Piece\$member to predicate p \in Piece;
%prenex;
%equality substitute q\_\_0;
%equality substitute rem;
%equality substitute r!;
%equality substitute p;
%with disabled (inRange) rewrite;
%% not expanding the ranges here is helpful to see the contradiction
%% y \in LOC \upto (\negate 1 + (LOC + SIZE)) \land
%% \lnot y \in LOC \upto (\negate 1 + (s? + LOC))
```

```
%%
%% the second range is within the first either when s? = 0 \lor s? \leq SIZE.
%% given the hyp \lnot LOC = s? + LOC, s? \neq 0.
%%
%split s? = SIZE;
%with disabled (inRange) rewrite;
%split s? = 0;
%rewrite;
next;
\end{zproofold}

\begin{zproofold}[dlDISPOSE1JoinNoAbuttV3]
split abt2 = \{\};
cases;
invoke DISPOSE1Join;
rearrange;
equality substitute join2;
rewrite;
next;
rewrite;
prenex;
rewrite;
use dlDISPOSE1AbtValue;
use dlDISPOSE1FrValue;
use dlDISPOSE1P2Value;
rearrange;
apply dlInFree1;
apply Piece\$member to predicate p2 \in Piece;
simplify;
rearrange;
% just for rearrangement
split DISPOSE1Join;
simplify;
apply extensionality to predicate abt2 = \{\};
prenex;
rewrite;
equality substitute join2;
equality substitute abt2;
rewrite;
split x = p2;
rewrite;
split x = r;
rewrite;
apply dlInAbutt;
rewrite;
rearrange;
cases;
% doesn't abutt to the right
apply dlInFuse;
rewrite;

apply extensionality to predicate \{p2 \} \cup \{q\_\_0: Piece | q\_\_0 \in fr \land p2 \abutt q\_\_0 \} = \{p2 \};
prenex;
rewrite;
split x = p2;
rewrite;
apply dlLocsCapEmpty ;
rewrite;
instantiate t==k;
instantiate t==r;
rewrite;
apply extensionality to predicate \locsOf k \cap \locsOf p2 = \{\};
apply extensionality to predicate \locsOf p2 \cap \locsOf r = \{\};
rewrite;
apply dlLocsOfCapEmpty;
rewrite;
rewrite;
split p2 . LOC + p2 . SIZE = k.LOC;
simplify;
cases;
rearrange;
simplify;
apply Piece\$member;
prenex;
rearrange;
```

```
equality substitute r;
equality substitute k;
rewrite;
rearrange;
split LOC\_\_0 = LOC;
rewrite;
cases;
rearrange;
equality substitute LOC\_\_0;
rewrite;
invoke DISPOSE1Join;
equality substitute join2;
equality substitute abt2;
rewrite;
apply extensionality to predicate join2 = \{p2\};
prenex;
rearrange;
rewrite;
with normalization rewrite;
rewrite;
split r = x;
rewrite;
prenex;
rewrite;
instantiate x\_\_0 == x;
rearrange;
prenex;
rewrite;

next;
\end{zproofold}
\begin{zproofold}[dlDISPOSE1JoinNoAbuttV3]
apply dlLocsCapEmpty;
apply extensionality;
prove;
invoke DISPOSE1Join;
rearrange;
equality substitute join2;
rewrite;
with normalization rewrite;
equality substitute abt2;
%NOTE-1: why is it x can't be p2? I STILL CAN'T SEE THE CONTRADICTION
%with predicate (x \in \{q\_\_5: Piece | q\_\_5 \in fr \land p2 \abutt q\_\_5 \}) rewrite;
%with predicate (r \in \{q\_\_4: Piece | q\_\_4 \in fr \land p2 \abutt q\_\_4 \}) rewrite;
rewrite;
rearrange;
% use the precondition
instantiate t == r;
rewrite;
% NOTE-1: that's a kind of wild-guess out of no other option
instantiate x\_\_0 == x.LOC;
apply dlLocsOfProp to expression \locsOf~p2;
apply dlLocsOfProp to expression \locsOf~r;
rewrite;
apply dlInAbutt to predicate p2 \abutt x;
apply dlInAbutt to predicate p2 \abutt r;
apply dlInFuse;
rewrite;
% use the invariant of Free1;
apply dlInFree1;
apply dlInInvFree1Sep;
apply dlPieceBefore;
% so x can't abutt p2 then, why not? well... either it's r, which is not the case
split x = r;
rewrite;
% or it's before r
split x.LOC < r.LOC;
cases;
rearrange;
instantiate p1 == x, p2\_\_0 == r;
rearrange;
rewrite;
instantiate p1 == r, p2\_\_0 == x;
apply dlInAbutt to predicate p2 \abutt r;
apply dlInAbutt to predicate p2 \abutt x;
```

```
rewrite;
next;
% MAKE A LEMMA ABOUT abt2 TO CLARIFY WHERE THE CONTRADICTION IS
\end{zproofold}
```

References

Bibliography

[1] Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Sci. Comput. Program.*, 74(4):219–237, 2009.

[2] Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Science of Computer Programming*, 74(4):219–237, 2009. cited By (since 1996) 10.

[3] Leo Freitas. Proving theorems with z/eves. Technical report, University of York, 2004.

[4] Leo Freitas and Jim Woodcock. Mechanising mondex with z/eves. *Formal Aspects of Computing*, 20(1):117–139, 2008. cited By (since 1996) 11.

[5] Leo Freitas and Jim Woodcock. A chain datatype in z. *International Journal of Software and Informatics*, 3(2-3):357–374, 2009.

[6] C. B. Jones and R. C. F. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990.

[7] Cliff Jones, Peter O'Hearn, and Jim Woodcock. Verified software: a grand challenge. *IEEE Computer*, 39(4):93–95, 2006.

[8] Jim Woodcock and Leo Freitas. Linking VDM and Z. In *International Conference on Engineering of Complex Computer Systems*, pages 143–152, Belfast, 2008. cited By (since 1996) 0; Conference of 13th IEEE International Conference on the Engineering of Complex Computer Systems, ICECCS 2008; Conference Date: 31 March 2008 through 4 April 2008; Conference Code: 72055.