Calculational Derivation of Algorithms on Tree-based Pointer Structures

Michael Butler
Dept. of Electronics & Computer Science, Univ. of Southampton,
Southampton SO17 1BJ, United Kingdom
M.J.Butler@ecs.soton.ac.uk

Abstract

We describe an approach to the derivation of correct algorithms on tree-based pointer structures. The approach is based on identifying abstract representations of commonly used pointer manipulations on trees and providing calculational-style refinement rules for these manipulations. This allows the algorithmic structure of a program to be derived using abstract data representations before introducing pointer structures.

1 Introduction

The derivation of correct algorithms involving pointers is difficult. In particular, the potentially complex connections through data paths within pointer structures make it difficult to reason about the side-effects of pointer assignment. One promising approach to dealing with this problem is described by Möller [9]. There, the store is an explicit program variable represented by a partial map. An algebra of partial maps is used to transform programs involving abstract data types such as lists into programs involving pointer structures. The derivation follows a standard data refinement approach using a function that extracts the appropriate abstract data element from the store and the algebra of partial maps to control side-effects.

The example derivations presented in [9] only involve algorithms on linked-list structures. We tried out the approach in the derivation of algorithms on tree structures and found it to be reasonably successful. Although the derivations were fairly mechanical, they were long and detailed, and therefore tedious and error prone. One reason for this was that the overhead of dealing with side-effects was quite considerable for tree structures. As an example of such overhead, consider the case of inserting an element in a binary search tree. While descending the tree searching for the correct place in which to insert the element, we used an invariant which stated that the entire search tree, excluding the sub-tree whose root is the node we are currently looking at, must remain unchanged. This invariant had to be shown to be preserved by each step of the descent. Such an overhead seems excessive considering the only assignments used in a descent step were `p:=p^.left` or `p:=p^.right` which don't even change the store. A second cause of the lengthy and detailed derivations was that, as well as changing the data representation, they also involved changing from an applicative style of specification (i.e., functional programs) to an imperative style. Algorithms on the abstract tree type are more naturally described in an applicative fashion, while algorithms involving pointer structures are more naturally described in an imperative fashion.

In this paper, we describe an approach that tries to factor out and simplify some of these difficulties. We enrich our abstract trees in a way that allows us to describe algorithms on them in an imperative fashion. Thus we can develop the necessary imperative structure before introducing the pointer data structures. Abstract trees are enriched in two ways: with paths and with the $*$-tree (or arbitrary tree). Paths are used to access subtrees and, instead of descending a tree in a recursive manner, we simply extend a path giving a deeper subtree. Abstract paths are represented at the concrete level by pointers. The $*$-tree provides a simple way of dealing with a certain form of sharing that arises during manipulation of structures. We describe some operations on our enriched trees and provide rules for transforming these into simple pointer operations. Much of the overhead of dealing with side effects is taken care of by these transformation rules. Although the rules presented here are only intended for the derivation of algorithms on tree-based pointer structures, the general approach is applicable to other data structures.

The program derivation framework used in our approach is the *refinement calculus* of Back, Morgan and Morris [1, 10, 11]. The refinement calculus is a formalisation of the stepwise refinement method of program construction. The required behaviour of the program is specified as an abstract, possibly non-executable, program which is then refined by a series of correctness-preserving transformations into an efficient, executable program.

Sections 2 and 5 introduce the required parts of the refinement calculus. Section 3 describes some operations on abstract trees. Section 4 describes our model of pointer structures. Section 6 describes the relation between abstract trees and pointer structures and Section 7 presents a collection of laws for transforming abstract-tree operations to pointer operations. Section 8 applies these laws to some examples, and Section 9 provides some general guidelines on their application.

2 Program Derivation Framework

Specification and Programming Notation The programming notation used is an extension of Dijkstra's guarded command language [4]. As well as containing assignments statements ($x := E$), sequential composition ($S;\ T$), IF-statements (**if** G **then** S **else** T **fi**), DO-loops (**do** $G \rightarrow S$ **od**), **skip**, and **abort**, the extended language includes the following constructs:

- *Magic:* The statement **magic** satisfies every program specification but is unimplementable.
- *Assertion:* The statement $\{p\}$, where p is a predicate stating a condition on the program variables, behaves as **skip** if p is satisfied, otherwise it behaves as **abort**.
- *Coercion:* The statement $[p]$, where p is a predicate stating a condition on the program variables, behaves as **skip** if p is satisfied, otherwise it behaves as **magic**. Note that we usually omit the ';' when $\{p\}$ and $[p]$ are placed before and after other statements.
- *Non-deterministic Assignment:* The statement $x := x' \mid P$, where P is a predicate relating x and x', assigns a value x' satisfying P to program variable x. $x := x' \mid P$ behaves as **magic** if P cannot be satisfied.
- *Local Variables:* The statement $|[\mathbf{var}\ y \cdot S]|$, behaves as S except that the program variable y is local to S.
- *Procedures and Parameters:* We declare a (parameterless) procedure M, by declaring **proc** $M \mathrel{\hat{=}} S$, where S is some statement. The effect of a call to M is the same as simply substituting program statement S for M. Program variables and substitution are used as a parameter-passing mechanism:
  - *Substitution by Value:* In the statement $S[\mathbf{val}\ x \backslash e]$, program variable x is initialised to the value e, and its final value is ignored.
  - *Substitution by Result:* In the statement $S[\mathbf{res}\ x \backslash y]$, the final value of program variable x is assigned to program variable y.
  - *Substitution by Value/Result:* In the statement $S[\mathbf{valres}\ x \backslash y]$, program variable x is initialised to the value of program variable y, and the final value of x is assigned to y.

  Parameters are added to a procedure by declaring, for example,
  proc $M(\mathbf{val}\ x) \mathrel{\hat{=}} S$, and the call $M(e)$ is then short for $S[\mathbf{val}\ x \backslash e]$.
- *Recursion:* A recursive statement is written $(\mu X \cdot F\ X)$, where $F(X)$ is some statement containing statement X. The declaration **proc** $M \mathrel{\hat{=}} F(M)$ is short for **proc** $M \mathrel{\hat{=}} (\mu X \cdot F\ X)$, and, for example, **proc** $M(\mathbf{val}\ x) \mathrel{\hat{=}} F(M(e))$ is short for **proc** $M \mathrel{\hat{=}} (\mu X \cdot F(X[\mathbf{val}\ x \backslash e]))$.
- *Simultaneous execution:* For statements S_1 and S_2 which assign to separate variables, $S_1 \parallel S_2$ is the simultaneous execution of S_1 and S_2, e.g.,
  $x := e \ \parallel\ y := f \quad = \quad x, y := e, f.$

A weakest-precondition definition of simultaneous execution is given in [2]. Weakest-precondition definitions of all the other extensions can be found in [1, 10, 11].

Refinement Statement T is said to refine statement S, written $S \leq T$, if [1, 10, 11]

$$wp(S, q) \quad \Rightarrow \quad wp(T, q), \quad \text{for all postconditions } q,$$

where wp is Dijkstra's weakest-precondition predicate transformer. The refinement ordering models the notion of total-correctness preserving program refinement. A total-correctness specification is typically given as a precondition-postcondition pair $(pre, post)$, and program S satisfies $(pre, post)$ if $pre \Rightarrow wp(S, post)$. Now, for programs S and T, $S \leq T$ holds if and only if T satisfies any specification satisfied by S.

In the refinement calculus, specifications are typically represented as abstract programs of the form $\{pre\}\ x := x' \mid post$ (the *specification statement*). The calculus then provides laws for transforming abstract program structures into code, i.e., programs involving only implementable constructs such as deterministic assignment, sequential composition, IF-statements, and DO-statements. For example, the following law is used to transform a specification statement into a deterministic assignment:

$$\frac{p \ \Rightarrow \ Q[x \backslash e]}{\{p\}\ x := x' \mid Q \quad \leq \quad x := e}$$

Stepwise development is supported since the refinement relation is transitive, and piecewise development is supported since the program constructors preserve refinement, e.g., $S_1 \leq S_2 \ \Rightarrow \ (T;\ S_1) \leq (T;\ S_2)$.

3 Trees

Assume that *Item* is some given type, then a (binary) tree of *Item* is defined as a recursive data type:

$$Tree \quad \hat{=} \quad \epsilon \ \mid\ * \ \mid\ \mathsf{node}(Item \times Tree \times Tree).$$

That is, a tree can be the empty-tree, the $*$-tree, or a node consisting of an item and a left and a right subtree. The role of the $*$-tree will become clear later. For the moment we can think of it as being similar to the empty tree. The components of a node tree are accessed as follows: for $L, R : Tree$, $b : Item$,

$$\begin{array}{rcl} roottree\ \mathsf{node}(b, L, R) & \hat{=} & b \\ lefttree\ \mathsf{node}(b, L, R) & \hat{=} & L \\ righttree\ \mathsf{node}(b, L, R) & \hat{=} & R. \end{array}$$

A procedure that checks if an item b is in tree t is specified as follows:

$$\begin{array}{l} \textbf{proc}\ Member(\textbf{val}\ t : Tree;\ \ b : Item;\ \textbf{res}\ f :\ \textbf{Bool}\ ) \ \hat{=} \\ \quad \textbf{if}\ b \in elems(t)\ \textbf{then}\ f := true\ \textbf{else}\ f := false. \end{array}$$

Here *elems* is defined recursively in the usual way. Now assume that t is a binary search tree, with items ordered by $\preceq$, and that t contains no occurrences of $*$. Using laws of the refinement calculus, this procedure may be refined to the recursive procedure of Figure 1. We omit this and other derivations of algorithms on abstract tree structures. Examples of similar derivations may be found in [10]. Our main concern will be the refinement of algorithms on abstract trees into algorithms on pointer structures.

A path is a sequence of left or right moves leading to a subtree:

$$Path \quad \hat{=} \quad seq(\mathsf{left} \mid \mathsf{right}).$$

The subtree of t reached by following path m is denoted t/m which is (partially) defined as follows: for $t, L, R : Tree$, $b : Item$, $m : Path$,

$$\begin{array}{rcl} t\ /\ \langle\rangle & \hat{=} & t \\ \mathsf{node}(b, L, R)\ /\ \langle \mathsf{left} \rangle ^\frown m & \hat{=} & L/m \\ \mathsf{node}(b, L, R)\ /\ \langle \mathsf{right} \rangle ^\frown m & \hat{=} & R/m. \end{array}$$

$$\begin{array}{l}
\textbf{proc } \mathit{Member}(\textbf{val } t : \mathit{Tree};\ \ b : \mathit{Item};\ \textbf{res } f :\ \textbf{Bool}\ ) \mathrel{\hat{=}} \\
\textbf{if } t = \epsilon \textbf{ then } f := \mathit{false} \\
\textbf{else if } \mathit{roottree}\ t = b \textbf{ then } f := \mathit{true} \\
\textbf{else if } \mathit{roottree}\ t \succ b \textbf{ then } \mathit{Member}(\mathit{lefttree}\ t, b, f) \\
\textbf{else } \mathit{Member}(\mathit{righttree}\ t, b, f) \textbf{ fi fi fi}
\end{array}$$

Figure 1: Member procedure on abstract trees

$$\begin{array}{l}
\textbf{proc } \mathit{Insert}(\textbf{valres } t : \mathit{Tree};\ \textbf{val } b : \mathit{Item}) \mathrel{\hat{=}} \\
|[\textbf{var } m_t : \mathit{Path} \cdot \\
m_t := \langle\rangle; \\
\textbf{do } t/m_t \neq \epsilon\ \wedge\ \mathit{roottree}(t/m_t) \neq b \rightarrow \\
\qquad \textbf{if } \mathit{roottree}(t/m_t) \succ b \textbf{ then } m_t := m_t \char`\^ \langle \mathsf{left} \rangle \\
\qquad \textbf{else } m_t := m_t \char`\^ \langle \mathsf{right} \rangle \textbf{ fi} \\
\textbf{od}; \\
\textbf{if } t/m_t = \epsilon \textbf{ then } t := t[m_t \backslash \mathsf{node}(b, \epsilon, \epsilon)]\ \ \textbf{fi}\ \ ]|
\end{array}$$

Figure 2: Insert procedure on abstract trees

The set of all paths of a tree is defined as follows: for $L, R : \mathit{Tree}$, $b : \mathit{Item}$,

$$\begin{array}{lll}
\mathit{paths}\ \ \epsilon & \hat{=} & \{\ \langle\rangle\ \} \\
\mathit{paths}\ \ * & \hat{=} & \{\ \langle\rangle\ \} \\
\mathit{paths}\ \ \mathsf{node}(b, L, R) & \hat{=} & \{\ \langle \mathsf{left} \rangle \char`\^ m \mid m \in (\mathit{paths}\ L)\ \} \cup \\
 & & \{\ \langle \mathsf{right} \rangle \char`\^ m \mid m \in (\mathit{paths}\ R)\ \}.
\end{array}$$

Given trees t, u, and path m, $t[m\backslash u]$ represents the tree t with the subtree reached using m replaced by u: for $t, u, L, R : \mathit{Tree}$, $b : \mathit{Item}$, $m : \mathit{Path}$,

$$\begin{array}{rlll}
t & [\ \langle\rangle \backslash u\ ] & \hat{=} & u \\
\mathsf{node}(b, L, R) & [\ \langle \mathsf{left} \rangle \char`\^ m \backslash u\ ] & \hat{=} & \mathsf{node}(b,\ L[m\backslash u],\ R) \\
\mathsf{node}(b, L, R) & [\ \langle \mathsf{right} \rangle \char`\^ m \backslash u\ ] & \hat{=} & \mathsf{node}(b,\ L,\ R[m\backslash u]).
\end{array}$$

We have that

$$m \in \mathit{paths}(t) \quad \Rightarrow \quad (t[m\backslash u])/m = u, \quad t[m\backslash(t/m)] = t, \quad (t[m\backslash u])[m\backslash v] = t[m\backslash v].$$

The *Insert* procedure of Figure 2 inserts an item in a tree, preserving the binary search order. Here, the path m_t is continually extended in the appropriate direction until a node is reached which contains the item b, in which case the tree is left unchanged, or until the empty tree is reached, in which case a new node containing b is added to t at the place determined by m_t.

In the *Insert* procedure, our use of $t[m\backslash u]$ is a form of *paste* operation where we replace an empty subtree with some other tree. Another special case of replacement is the *cut* operator which replaces a sub-tree of t with the $*$-tree:

$$t \restriction m \quad \hat{=} \quad t[m\backslash *].$$

We have that

$$m \in \mathit{paths}(t) \quad \Rightarrow \quad (t \restriction m)[m\backslash u] = t[m\backslash u], \quad (t \restriction m)[m\backslash(t/m)] = t.$$

The second equality above shows that a tree t is cut in two by

$$t, u := t \restriction m, t/m.$$

4 Pointer Structures

In order to implement binary trees in an imperative programming language, we would typically use pointer structures defined by the following types:

```
TYPE  TreePtr = POINTER TO Node;
      Node = RECORD
                       root:Item;
                       left,right:TreePtr
                     END;
```

In the style of [9], a structure of this type would be modelled explicitly as a partial map from pointers to records:

$$\mathit{TreeStore} \quad \hat{=} \quad \mathit{TreePtr} \nrightarrow \mathit{Node}.$$

Here we assume that $\mathit{TreePtr}$ is some infinite type, and that Node is a record type with the structure described above. The components of a node record r are accessed by $r.\mathit{root}$, etc, and a record is constructed by $\mathit{mk\_node}(b, pl, pr)$ for $b : \mathit{Item}, pl, pr : \mathit{TreePtr}$.

Indirect access of a value via a pointer is modelled as a lookup in the tree store. Let $\sigma : \mathit{TreeStore}$ be a store, and let pt be a pointer to a record in this store, i.e., $pt \in \operatorname{dom} \sigma$. Then a reference to $pt\hat{}$ would represent the record $(\sigma\ pt)$, and a reference to $pt\hat{}.\mathit{root}$ would represent the item $(\sigma\ pt).\mathit{root}$.

Indirect assignment via a pointer is modelled by an update of the store, e.g., $pt\hat{}.\mathit{root} := b$ would be modelled by ($f \oplus g$ denotes map f overridden by map g):

$$\sigma := \sigma \oplus \{\ pt \mapsto \mathit{mk\_node}(b,\ (\sigma\ pt).\mathit{left},\ (\sigma\ pt).\mathit{right})\ \}.$$

We shall write this as $(\sigma\ pt).\mathit{root} := b$.

The **new** statement that allocates space for a new record in the store is modelled by the following procedure:

$$\begin{array}{l} \textbf{proc}\ \mathit{new\_node}(\textbf{res}\ pt;\ \ \textbf{valres}\ \sigma)\ \hat{=} \\ pt, \sigma := pt', \sigma' \mid pt' \notin \operatorname{dom}\sigma \wedge \exists\, b, L, R \cdot \sigma' = \sigma \cup \{pt' \mapsto \mathit{mk\_node}(b, L, R)\}. \end{array}$$

For $\sigma : \mathit{TreeStore}$, let $\mathit{left}\ \sigma$ and $\mathit{right}\ \sigma$ be defined as follows:

$$\begin{array}{rcl} \mathit{left}\ \sigma & \hat{=} & \{\ pt \mapsto (\sigma\ pt).\mathit{left} \mid pt \in \operatorname{dom}\sigma\ \} \\ \mathit{right}\ \sigma & \hat{=} & \{\ pt \mapsto (\sigma\ pt).\mathit{right} \mid pt \in \operatorname{dom}\sigma\ \}. \end{array}$$

The relation $(\mathit{left}\ \sigma) \cup (\mathit{right}\ \sigma)$ then models the graph of interconnections in the store σ.

We assume that the value nil is of type $\mathit{TreePtr}$, but that nil is never in the domain of σ so that access via nil is unsafe. We say that a pointer is well-defined (w.r.t. σ) if it is in the domain of σ or if it equals nil. We assume that all pointers within σ are well-defined. These assumptions on σ are represented by:

$$\begin{array}{rcl} \mathit{WellFormed}\ \sigma & \hat{=} & \mathsf{nil} \notin \operatorname{dom}\sigma\ \wedge \\ & & \operatorname{ran}(\mathit{left}\ \sigma)\ \subseteq\ \{\mathsf{nil}\} \cup (\operatorname{dom}\sigma)\ \wedge \\ & & \operatorname{ran}(\mathit{right}\ \sigma)\ \subseteq\ \{\mathsf{nil}\} \cup (\operatorname{dom}\sigma). \end{array}$$

The assumption that all pointers be well-defined is rather strong considering that $\mathit{new\_node}$ assigns arbitrary values to the fields of the record that it creates. However, in the refinement rules that we shall present, calls to $\mathit{new\_node}$ will always be followed immediately by assignments to the fields of the new record.

The implementation of abstract trees using pointer structures is a form of *data refinement*, where abstract program variables are replaced by concrete state variables using some abstraction relation. Section 6 will describe the abstraction relation between trees and pointer structures. Section 7 will describe some laws for data-refining operations on trees, such as those in Section 3, into operations on pointer structures. These laws show how basic operations on trees such as $x := \mathit{roottree}(t/m_t)$, $m_t := m_t\hat{}\langle\mathsf{left}\rangle$, $t := \mathsf{node}(b, \epsilon, \epsilon)$, and $t, u := t \upharpoonright m_t, t/m_t$, may be data-refined directly into pointer operations. Before looking at the data-refinement of trees, the required background on data refinement is introduced in the next section.

Law 1 $\dfrac{p \wedge R \wedge Q \;\Rightarrow\; (\exists\, a' \cdot R[a,c,u \backslash a',c',u'] \wedge P)}{\mathcal{D}(\{p\}\; a,u := a',u' \mid P) \;\;\leq\;\; c,u := c',u' \mid Q}$

Law 2 $\mathcal{D}(S_1;\, S_2) \;\;\leq\;\; \mathcal{D}(S_1);\, \mathcal{D}(S_2)$

Law 3 $\dfrac{R \;\;\Rightarrow\;\; G \Leftrightarrow H}{\mathcal{D}(\mathbf{if}\; G\; \mathbf{then}\; S_1\; \mathbf{else}\; S_2\; \mathbf{fi}) \;\;\leq\;\; \mathbf{if}\; H\; \mathbf{then}\; \mathcal{D}(S_1)\; \mathbf{else}\; \mathcal{D}(S_2)\; \mathbf{fi}}$

Law 4 $\dfrac{R \;\;\Rightarrow\;\; G \Leftrightarrow H}{\mathcal{D}(\mathbf{do}\; G \rightarrow S\; \mathbf{od}) \;\;\leq\;\; \mathbf{do}\; H \rightarrow \mathcal{D}(S)\; \mathbf{od}}$

Law 5 $\dfrac{\mathcal{D}(X) \leq Y \;\;\Rightarrow\;\; \mathcal{D}(F\; X) \leq G(Y)}{\mathcal{D}(\mu\, X \cdot F\; X) \;\;\leq\;\; \mu\, Y \cdot G\; Y}$

Law 6 $\dfrac{p \wedge R \;\Rightarrow\; R[a,c,u \backslash e,f,g]}{\mathcal{D}(\{p\}\; S[\mathbf{val}\; a,u \backslash e,g]) \;\;\leq\;\; (\mathcal{D}(S))[\mathbf{val}\; c,u \backslash f,g]}$

Law 7 $\mathcal{D}[\![a',c',R']\!](S[\mathbf{res}\; a,u \backslash a',u']) \;\;\leq$
$\quad (\mathcal{D}[\![a,c,R]\!](S))[\mathbf{val}\; c,u \backslash c',u']$

Law 8 $\mathcal{D}[\![a',c',R']\!](S[\mathbf{valres}\; a,u \backslash a',u']) \;\;\leq$
$\quad (\mathcal{D}[\![a,c,R]\!](S))[\mathbf{valres}\; c,u \backslash c',u']$

Law 9 $\mathcal{D}[\![a_1,c_1,R]\!](|[\mathbf{var}\; a_2 \cdot S]|) \;\;\leq$
$\quad |[\mathbf{var}\; c_2 \cdot \mathcal{D}[\![(a_1,a_2),(c_1,c_2),R \wedge Q]\!](S)]|$

Figure 3: Data Refinement Laws

5 Data Refinement

Data refinement involves replacing abstract program variables with concrete program variables using an abstraction relation. Let S be a statement with program variables u, a, and let T be a statement with variables u, c, then S is data-refined by T under abstraction relation R if the following holds, for all postconditions q where c **nfi** q [1, 10, 11, 5] (x **nfi** P means that x is not free in P):

$$R \wedge wp(S,q) \;\;\Rightarrow\;\; wp(T, \exists\, a \cdot R \wedge q).$$

This paper makes use of the *least data-refinement* of a statement. Again, let S be a statement with program variables u, a, and let R be an abstraction relation relating a and c, then the least refined statement on program variables u, c which is also a data refinement of S under R is denoted:

$$\mathcal{D}[\![a,c,R]\!](S).$$

We calculate a data refinement of a statement S under R by calculating a refinement of $\mathcal{D}[\![a,c,R]\!](S)$. Laws that support such calculation by distributing $\mathcal{D}[\![a,c,R]\!]$ through the structure of S may be found in [6, 12, 13]. Some of these law are repeated in Figure 3. (In this figure, $\mathcal{D}[\![a,c,R]\!]$ has been shortened to $\mathcal{D}$ in places.)

Decomposing Data Refinement Often it is convenient to break a data refinement step into an *augmentation* step, which involves introducing concrete variables without removing the abstract ones, and an *diminution* step which involves eliminating the abstract variables.

Law 10 $\mathcal{A}[\![c, R]\!](\{p\}) \quad \leq \quad \{p\}$

Law 11 $\mathcal{A}[\![c, R]\!](S) \quad \leq \quad \{R\}\ \mathcal{A}[\![c, R]\!](S)$

Law 12 $\mathcal{A}[\![c, R \wedge Q]\!](S) \quad \leq \quad \mathcal{A}[\![c, R]\!](S)\ [Q]$

Law 13 $\mathcal{A}[\![(c_1, c_2), R]\!](S) \quad \leq \quad \mathcal{A}[\![c_1, R]\!](S)$ *provided* c_2 **nfi** R

Figure 4: Data Augmentation Laws

Augmentation is a special case of general data refinement where the list of abstract variables to be replaced is empty, and we denote the least augmentation of S with c under R by

$$\mathcal{A}[\![c, R]\!](S).$$

Some laws associated with augmentation are presented in Figure 4. By preserving the abstract variables in an augmentation step, we can introduce the refinement relation as an explicit assertion (see Law 11), and make use of it in further refinement steps. In particular, it allows us to modularise data refinement in the following manner:

Suppose we know that $\mathcal{A}[\![c_1, R]\!](S) \leq T$, but we are using S in a context where c_2 is also being added and we require abstraction relation Q to hold. Thus we require a refinement of $\mathcal{A}[\![(c_1, c_2), R \wedge Q]\!](S)$, so we proceed as follows:

$$\begin{array}{ll}
 & \mathcal{A}[\![(c_1, c_2), R \wedge Q]\!](S) \\
\leq & \qquad \text{“\textit{by Law 11}”} \\
 & \{R \wedge Q\}\ \ \mathcal{A}[\![(c_1, c_2), R \wedge Q]\!](S) \\
\leq & \qquad \text{“\textit{by Law 12}”} \\
 & \{R \wedge Q\}\ \ \mathcal{A}[\![(c_1, c_2), R]\!](S)\ \ [Q] \\
\leq & \qquad \text{“\textit{by Law 13,} } c_2 \ \mathbf{nfi}\ R\text{”} \\
 & \{R \wedge Q\}\ \ \mathcal{A}[\![c_1, R]\!](S)\ \ [Q] \\
\leq & \qquad \text{“\textit{assumption:} } \mathcal{A}[\![c_1, R]\!](S) \leq T\text{”} \\
 & \{R \wedge Q\}\ \ T\ \ [Q].
\end{array}$$

The next step is to remove the coercion $[Q]$ (assertions can always be removed since $\{p\} \leq$ **skip** for any p). Showing that

$$\{R \wedge Q\}\ T\ [Q] \quad \leq \quad T$$

is called a *non-interference* proof, since it tells us that execution of T preserves, or does not interfere with, the invariant Q. In the case where T only makes assignments to variables not mentioned by Q, then Q is trivially preserved by T, as, for example, in the following law about assignment:

Law 14 $\qquad \{Q\}\ \ x := e\ \ [Q] \quad \leq \quad x := e,$ $\qquad$ provided x **nfi** Q.

This law does not apply in the case that Q mentions x, but later we provide other non-interference laws which will allow certain coercions to be removed in tree refinement.

Diminution is also a special case of general data refinement where the list of variables being introduced is empty. Usually, diminution is checked syntactically and does not require any abstraction relation or proof. In order to eliminate abstract variable a from a program, we firstly refine the program so that all guards are independent of a, and all assignments to variables other than a are independent of a. All assignments to a are then removed.

6 Refinement of Trees

Each abstract tree t_i will be represented at the concrete level by a pointer pt_i and a store σ; a single store σ will be shared by each tree t_i. A path will always be associated with a single tree so path variables are always subscripted by the name of the tree with which they are associated, e.g., path m_t is associated with tree t. Rather than representing a path m_t directly at the concrete level, we instead represent the subtree t/m_t with a pointer pm_t. We will allow more than one pointer to (possibly) different parts of the same tree structure.

Assume that t and u are distinct program variables of type *Tree*. We would expect that if we make an assignment to u, then the value of t remains unchanged. But suppose we firstly make the assignment $u := t/m_t$. This could be implemented by the assignment $pu := pm_t$. Unfortunately, any implementation of an assignment to u following $pu := pm_t$ would have the side-effect of changing the concrete representation of t.

One solution to this problem would be to require that different abstract trees be represented by disjoint graphs in the store. In that case, the assignment $u := t/m_t$ would have to be implemented by creating a copy of t/m_t in the store. But we want to avoid unnecessary copying.

We shall see that $pu := pm_t$ is a useful assignment for efficient manipulation of trees in certain cases, so we want to be able to represent it at the abstract level. Our solution involves the use of the cut operator described earlier. We regard $pu := pm_t$ as an implementation of

$$t, u := t \restriction m_t, t/m_t.$$

We shall take the view that the $*$-tree may be represented by any pointer structure, so that a representation of the $*$-tree may be changed in any way without violating the abstraction relation. This may be seen more clearly in the definition of $Match(t, pt, \sigma)$ which is true if and only if the pointer structure in σ starting at pt is a valid representation of abstract t: for $L, R : Tree$, $b : Item$, $pt : TreePtr$, $\sigma : TreeStore$,

$$\begin{array}{llllll} Match & \epsilon & pt & \sigma & \hat{=} & (pt = \mathsf{nil}) \\ Match & * & pt & \sigma & \hat{=} & true \\ Match & \mathsf{node}(b, L, R) & pt & \sigma & \hat{=} & pt \in \operatorname{dom} \sigma \ \wedge \ (\sigma\ pt).root = b \ \wedge \\ & & & & & \quad Match\ L\ ((\sigma\ pt).left)\ \sigma \ \wedge \ Match\ R\ ((\sigma\ pt).right)\ \sigma. \end{array}$$

Here the difference between ϵ and $*$ can be seen: $*$ will match with any structure, while ϵ will only match with nil.

Rather than requiring that different abstract trees be represented by disjoint graphs in the store, we only require that they be disjoint modulo $*$. The set of pointers in the representation of tree t starting at pt is given by $points(t, pt, \sigma)$ defined as follows: for $L, R : Tree$, $b : Item$, $pt : TreePtr$, $\sigma : TreeStore$,

$$\begin{array}{llllll} points & \epsilon & pt & \sigma & \hat{=} & \{\} \\ points & * & pt & \sigma & \hat{=} & \{\} \\ points & \mathsf{node}(b, L, R) & pt & \sigma & \hat{=} & \{pt\} \cup (points\ L\ ((\sigma\ pt).left)\ \sigma) \\ & & & & & \quad \cup\ (points\ R\ ((\sigma\ pt).right)\ \sigma). \end{array}$$

Here any pointers representing $*$ are ignored. Now the representations of trees t, u starting at pt, pu respectively are disjoint modulo $*$ iff $Disjoint(t, pt, u, pu, \sigma)$:

$$Disjoint(t, pt, u, pu, \sigma) \quad \hat{=} \quad (points\ t\ pt\ \sigma) \cap (points\ u\ pu\ \sigma) = \{\}.$$

The graph of a structure starting at pt representing tree t is the graph of σ restricted to the points representing t:

$$graph\ t\ pt\ \sigma \quad \hat{=} \quad (left\ \sigma \cup right\ \sigma)\ \cap\ ((points\ t\ pt\ \sigma) \times (points\ t\ pt\ \sigma)).$$

We shall require that such graphs be acyclic. If t does not contain $*$, and $Match(t, pt, \sigma)$ holds, then $graph(t, pt, \sigma)$ is guaranteed to be acyclic since it is the exact image of a tree. However it is possible for a tree containing $*$ to match a cyclic structure, e.g., consider $t = \mathsf{node}(b, \mathsf{node}(b, *, \epsilon), \epsilon)$ and $\sigma = \{ \ pt \mapsto mk\_node(b, pt, \mathsf{nil}) \ \}$. We will need to state explicitly that graphs be acyclic:

$$Acyclic(t, pt, \sigma) \quad \hat{=} \quad (graph\ t\ pt\ \sigma)^{+} \ \cap \ ID_{[TreePtr]} = \{\}.$$

That is, the transitive closure of the graph should not contain any identity mappings.

We can now state the abstraction invariant that is required to hold between tree t, pointer pt, and store σ:

$$AbsTree(t, pt, \sigma) \quad \hat{=} \quad WellFormed(\sigma) \ \wedge \ Match(t, pt, \sigma) \ \wedge \ Acyclic(t, pt, \sigma).$$

As stated earlier, rather than representing a path m_t directly, we represent the subtree t/m_t with a pointer pm_t. As part of the abstraction relation, we require the function *follow*, which returns the pointer reached by following path m_t starting at pointer pt: for $m : Path$, $pt : TreePtr$, $\sigma : TreeStore$,

$$\begin{array}{lllll} follow & pt & \langle\rangle & \sigma & \hat{=} & pt \\ follow & pt & \langle \mathsf{left} \rangle ^\frown m & \sigma & \hat{=} & follow\ ((\sigma\ pt).left)\ m\ \sigma \\ follow & pt & \langle \mathsf{right} \rangle ^\frown m & \sigma & \hat{=} & follow\ ((\sigma\ pt).right)\ m\ \sigma. \end{array}$$

It will be required that $pm_t = follow\ pt\ m_t\ \sigma$.

The pointer pm_t is not sufficient to allow us to implement a paste operation of the form $t := t[m_t \backslash u]$. In order to implement this operation, we need to change the parent node of pm_t in the tree hierarchy so that it points to the representation of u. For this, an extra pointer qm_t, modelling the parent of pm_t is introduced. We also need to know whether pm_t is the left or right descendant of qm_t, so we introduce a variable dm_t which can have the value left or right. In the case that pm_t has no parent, i.e., $m_t = \langle\rangle$, then dm_t will have the value root. We introduce:

$$Direction \ \hat{=} \ \mathsf{left} \mid \mathsf{right} \mid \mathsf{root}.$$

The abstraction relation associate with the representation of tree t and path m_t is then defined as follows: for $t : Tree$, $m_t : Path$, $pt, pm_t, qm_t, : TreePtr$, $dm_t : Direction$, $\sigma : TreeStore$,

$$\begin{array}{l} AbsTreePath(t, m_t, pt, pm_t, qm_t, dm_t, \sigma) \ \hat{=} \\ \quad AbsTree(t, pt, \sigma) \wedge \\ \quad m_t \in (paths\ t) \wedge \\ \quad follow\ pt\ m_t\ \sigma \ = \ pm_t \wedge \\ \quad (dm_t = \mathsf{root} \ \Rightarrow \ m_t = \langle\rangle \wedge qm_t = pm_t) \wedge \\ \quad (dm_t \neq \mathsf{root} \ \Rightarrow \ \ m_t \neq \langle\rangle \wedge (last\ m_t) = dm_t \wedge \\ \qquad\qquad\qquad\qquad follow\ pt\ (front\ m_t)\ \sigma = qm_t). \end{array}$$

Note that the pointers pm_t and qm_t are strongly bound up with the representation of t, whereas a path m_t is only bound to t by convention. However, the relation *AbsTreePath* forces a strong connection between m_t and t, and we shall only provide refinements of operations on m_t that preserve this relation.

7 Pointer Introduction Laws

In this section, we present a set of laws for transforming operations on abstract trees into operations on pointer structures.

7.1 Trees Without Paths

Let $Is\_node(t) \ \hat{=} \ t \neq \epsilon \wedge t \neq *$. The first law allows certain expressions involving trees to be replaced by expressions involving pointers:

Law 15

$$
\begin{array}{rcrcl}
\multicolumn{5}{l}{AbsTree(t,pt,\sigma) \;\Rightarrow} \\
t \neq * & \Rightarrow & t = \epsilon & \Leftrightarrow & pt = \mathsf{nil} \\
Is\_node(t) \;\wedge\; lefttree\ t \neq * & \Rightarrow & lefttree\ t = \epsilon & \Leftrightarrow & (\sigma\ pt).left = \mathsf{nil} \\
Is\_node(t) \;\wedge\; righttree\ t \neq * & \Rightarrow & righttree\ t = \epsilon & \Leftrightarrow & (\sigma\ pt).right = \mathsf{nil} \\
Is\_node(t) & \Rightarrow & roottree\ t & = & (\sigma\ pt).root.
\end{array}
$$

Using this law, and the data refinement distribution laws of Figure 3, we can derive a pointer implementation of the *Member* procedure. Let $T \mathrel{\hat{=}} AbsTree(t,pt,\sigma)$. We will calculate a refinement of $\mathcal{D}[\![T, t, (pt,\sigma)]\!](Member)$. Since *Member* is a recursive procedure, we start by assuming $\mathcal{D}[\![T, t, (pt,\sigma)]\!](Member) \leq Member'$ (see Law 5), and proceed as follows:

$$
\begin{array}{ll}
 & \mathcal{D}[\![T, t, (pt,\sigma)]\!](\\
 & \quad \mathbf{if}\ t = \epsilon\ \mathbf{then}\ f := \mathit{false} \\
 & \quad \mathbf{else\ if}\ roottree\ t = b\ \mathbf{then}\ f := \mathit{true} \\
 & \quad \mathbf{else\ if}\ roottree\ t \succ b \\
 & \quad \mathbf{then}\ Member(lefttree\ t,\ b,\ f) \\
 & \quad \mathbf{else}\ Member(righttree\ t,\ b,\ f) \\
 & \quad \mathbf{fi\ fi\ fi} \qquad\qquad\qquad\qquad) \\
\leq & \qquad \text{“ } \mathit{distribute}\ \mathcal{D},\ \mathit{transform\ guards\ using\ Law\ 15} \text{ ”} \\
 & \quad \mathbf{if}\ pt = \mathsf{nil} \\
 & \quad \mathbf{then}\ \mathcal{D}(f := \mathit{false}) \qquad\qquad (1) \\
 & \quad \mathbf{else\ if}\ (\sigma\ pt).root = b \\
 & \quad \mathbf{then}\ \mathcal{D}(f := \mathit{true}) \qquad\qquad (2) \\
 & \quad \mathbf{else\ if}\ (\sigma\ pt).root \succ b \\
 & \quad \mathbf{then}\ \mathcal{D}(Member(lefttree\ t,\ b,\ f)) \quad (3) \\
 & \quad \mathbf{else}\ \mathcal{D}(Member(righttree\ t,\ b,\ f)) \quad (4) \\
 & \quad \mathbf{fi\ fi\ fi}.
\end{array}
$$

$$
\begin{array}{ll}
(1) \leq & \qquad \text{“ } \mathit{trivial\ application\ of\ Law\ 1} \text{ ”} \\
 & f := \mathit{false}.
\end{array}
$$

Similarly, (2) $\leq f := true$. Before continuing with (3), we note the following law:

Law 16

$$
\begin{array}{l}
AbsTree(t,pt,\sigma) \wedge Is\_node(t) \;\Rightarrow \\
\quad AbsTree(lefttree\ t,\ (\sigma\ pt).left,\ \sigma) \;\wedge \\
\quad AbsTree(righttree\ t,\ (\sigma\ pt).right,\ \sigma).
\end{array}
$$

Proceed with (3), asserting the negated guard:

$$
\begin{array}{lll}
(3) & \leq & \mathcal{D}(\ \{t \neq \epsilon\}\ Member[\mathbf{val}\ t \backslash lefttree\ t][\mathbf{val}\ b \backslash b][\mathbf{res}\ f \backslash f]\ ) \\
 & \leq & \qquad \text{“ } \mathit{trivial\ distribution\ of}\ \mathcal{D}\ \mathit{through\ substitutions} \text{ ”} \\
 & & \mathcal{D}(\ \{t \neq \epsilon\}\ Member[\mathbf{val}\ t \backslash lefttree\ t]\ )\ [\mathbf{val}\ b \backslash b][\mathbf{res}\ f \backslash f] \\
 & \leq & \qquad \text{“ } \mathit{distribute}\ \mathcal{D}\ \mathit{assuming\ assertion} \text{ ”} \\
 & & \mathcal{D}(Member)\ [\mathbf{val}\ pt,\ \sigma \ \backslash\ (\sigma\ pt).left,\ \sigma][\mathbf{val}\ b \backslash b][\mathbf{res}\ f \backslash f] \\
 & \leq & \qquad \text{“ } \mathcal{D}(Member) \leq Member' \text{ ”} \\
 & & Member'\ [\mathbf{val}\ pt,\ \sigma \ \backslash\ (\sigma\ pt).left,\ \sigma][\mathbf{val}\ b \backslash b][\mathbf{res}\ f \backslash f] \\
 & = & Member'((\sigma\ pt).left,\ \sigma,\ b,\ f).
\end{array}
$$

proc $Member'$(**val** pt; σ; b; **res** f) $\hat{=}$
if $pt = \mathsf{nil}$ **then** $f := false$
else if $(\sigma\ pt).root = b$ **then** $f := true$
else if $(\sigma\ pt).root \succ b$ **then** $Member'((\sigma\ pt).left,\ \sigma,\ b,\ f)$
else $Member'((\sigma\ pt).right,\ \sigma,\ b,\ f)$ **fi fi fi**

Figure 5: Refined Member procedure

Similarly,

$$(4) \quad \leq \quad Member'((\sigma\ pt).right,\ \sigma,\ b,\ f).$$

The resulting refinement of *Member* is shown in Figure 5.

7.2 Trees With Paths

Certain terms involving trees and paths may be transformed into terms involving pointers using the following law:

Law 17

$$\begin{array}{rcrcl} \multicolumn{5}{l}{AbsTreePath(t, m_t, pt, pm_t, qm_t, dm_t, \sigma) \Rightarrow} \\ t/m_t \neq * & \Rightarrow & t/m_t = \epsilon & \Leftrightarrow & pm_t = \mathsf{nil} \\ Is\_node(t/m_t) \ \wedge\ lefttree(t/m_t) \neq * & \Rightarrow & lefttree\ (t/m_t) = \epsilon & \Leftrightarrow & (pm_t\ \sigma).left = \mathsf{nil} \\ Is\_node(t/m_t) \ \wedge\ righttree(t/m_t) \neq * & \Rightarrow & righttree\ (t/m_t) = \epsilon & \Leftrightarrow & (pm_t\ \sigma).right = \mathsf{nil} \\ Is\_node(t/m_t) & \Rightarrow & root\ (t/m_t) & = & (pm_t\ \sigma).root. \end{array}$$

In the remainder, we present laws describing refinements of assignments to trees or paths. These will be presented using the same format: for each law we present a list of abstract variables a_i, a list of concrete variables c_i, an abstraction invariant R_i, an abstract operation $AbsOp$ and a concrete operation $ConOp$. It will be the case that $\mathcal{A}[\![c_i, R_i]\!](AbsOp) \ \leq\ (AbsOp \parallel ConOp)$. Also, it will be the case that $ConOp$ is a diminution of $AbsOp \parallel ConOp$. The validity of each of the laws has been checked using Law 1 of Figure 3.

An operation for initialising a path to be empty is refined as follows:

Law 18 (Refinement of Initialise Path)

$$\begin{array}{rcl} a_{18} & \hat{=} & t, m_t \\ c_{18} & \hat{=} & pt, pm_t, qm_t, dm_t, \sigma \\ R_{18} & \hat{=} & AbsTreePath(t, m_t, pt, pm_t, qm_t, dm_t, \sigma) \\ AbsInitPath & \hat{=} & m_t := \langle\rangle \\ ConInitPath & \hat{=} & pm_t, qm_t, dm_t := pt, pt, \mathsf{root}. \end{array}$$

An operation for performing a left extension of a path is refined by updating the pointers pm_t, qm_t, and dm_t appropriately:

Law 19 (Refinement of Left Path)

$$\begin{array}{rcl} a_{19} & \hat{=} & t, m_t \\ c_{19} & \hat{=} & pt, pm_t, qm_t, dm_t, \sigma \\ R_{19} & \hat{=} & AbsTreePath(t, m_t, pt, pm_t, qm_t, dm_t, \sigma) \\ AbsLeftPath & \hat{=} & \{Is\_node(t/m_t)\}\ m_t := m_t\hat{\ }\langle\mathsf{left}\rangle \\ ConLeftPath & \hat{=} & pm_t, qm_t, dm_t := (pm_t\ \sigma).left, pm_t, \mathsf{left}. \end{array}$$

The refinement of a right move is defined symmetrically.

Copying of a path is refined using the following law:

Law 20 (Refinement of Copy Path)

$$
\begin{array}{rcl}
a_{20} & \hat{=} & t, m_t, n_t \\
c_{20} & \hat{=} & pt, pm_t, qm_t, dm_t, pn_t, qn_t, dn_t, \sigma \\
R_{20} & \hat{=} & AbsTreePath(t, m_t, pt, pm_t, qm_t, dm_t, \sigma) \wedge \\
 & & AbsTreePath(t, n_t, pt, pn_t, qn_t, dn_t, \sigma) \\
AbsCopyPath & \hat{=} & n_t := m_t \\
ConCopyPath & \hat{=} & pn_t, qn_t, dn_t := pm_t, qm_t, dm_t.
\end{array}
$$

7.3 Constructing New Nodes

Construction of a singleton node is implemented by calling *new_node* and making appropriate assignments to the fields of the new record:

Law 21 (Refinement of New Node)

$$
\begin{array}{rcl}
a_{21} & \hat{=} & t \\
c_{21} & \hat{=} & pt, \sigma \\
R_{21} & \hat{=} & AbsTree(t, pt, \sigma) \\
AbsNewNode & \hat{=} & t := \mathsf{node}(b, \epsilon, \epsilon) \\
ConNewNode & \hat{=} & new\_node(pt, \sigma); \\
 & & (\sigma\ pt).root, (\sigma\ pt).left, (\sigma\ pt).right := b, \mathsf{nil}, \mathsf{nil}.
\end{array}
$$

If we are using *AbsNewNode* in a context which includes another tree v, then we will want to ensure that $AbsNewNode \parallel ConNewNode$ does not interfere with the relation between v and its representation by pv, σ. Since *ConNewNode* updates σ, such non-interference cannot be checked syntactically. Instead, we present some laws to deal with this. Let DTV, V, and MV be defined as follows:

$$
\begin{array}{rcl}
DTV & \hat{=} & Disjoint(t, pt, v, pv, \sigma) \\
V & \hat{=} & AbsTree(v, pv, \sigma) \\
MV & \hat{=} & AbsTreePath(v, m_v, pv, pm_v, qm_v, dm_v, \sigma).
\end{array}
$$

$AbsNewNode \parallel ConNewNode$ preserves disjointness of t from any other tree v, and does not interfere with the abstraction relations for v or any path m_v associated with v:

Law 22 (Non-interference for New Node) Let $NewNode \mathrel{\hat{=}} AbsNewNode \parallel ConNewNode$.

$$
\begin{array}{rcl}
\{R_{21} \wedge DTV\}\ NewNode\ [DTV] & \leq & NewNode \\
\{R_{21} \wedge DTV \wedge V\}\ NewNode\ [V] & \leq & NewNode \\
\{R_{21} \wedge DTV \wedge MV\}\ NewNode\ [MV] & \leq & NewNode.
\end{array}
$$

The validity of this and remaining non-interference laws is checked using

$$
\frac{p \wedge q \ \Rightarrow\ q[x \backslash e]}{\{p \wedge q\}\ x := e\ [q] \ \leq\ x := e}
$$

7.4 Cutting and Pasting

The operation which cuts a tree in two parts is refined using the following rule:

Law 23 (Refinement of Cut)

$$\begin{array}{rcl}
a_{23} & \hat{=} & t, u, m_t \\
c_{23} & \hat{=} & pt, pu, pm_t, qm_t, dm_t, \sigma \\
R_{23} & \hat{=} & AbsTreePath(t, m_t, pt, pm_t, qm_t, dm_t, \sigma) \wedge \\
 & & AbsTree(u, pu, \sigma) \wedge \\
 & & Disjoint(t, pt, u, pu, \sigma) \\
AbsCut & \hat{=} & t, u := t \restriction m_t, t/m_t \\
ConCut & \hat{=} & pu := pm_t.
\end{array}$$

Since this operation changes tree t, it is possible for it to interfere with the abstraction relation for some path n_t which is different to m_t. Let NT be defined as:

$$NT \quad \hat{=} \quad AbsTreePath(t, n_t, pt, pn_t, qn_t, dn_t, \sigma).$$

The cut operation does not interfere with NT provided n_t does not reach into the subtree of t pointed to by m_t ($\preceq$ denotes the subsequence ordering):

Law 24 (Non-interference for Cut) Let $Cut \ \hat{=}\ AbsCut \parallel ConCut$.

$$\{R_{23} \wedge NT \wedge m_t \not\preceq n_t\}\ Cut\ [NT] \quad \leq \quad Cut.$$

The paste operation $t := t[m_t \backslash u]$ is used to compose trees t and u. To ensure that the representations of t and u remain disjoint, we only allow for the refinement of a paste operation where u becomes $*$:

Law 25 (Refinement of Paste)

$$\begin{array}{rcl}
a_{25} & \hat{=} & t, u, m_t \\
c_{25} & \hat{=} & pt, pu, pm_t, qm_t, dm_t, \sigma \\
R_{25} & \hat{=} & AbsTreePath(t, m_t, pt, pm_t, qm_t, dm_t, \sigma) \wedge \\
 & & AbsTree(u, pu, \sigma) \wedge \\
 & & Disjoint(t, pt, u, pu, \sigma) \\
AbsPaste & \hat{=} & t, u := t[m_t \backslash u], * \\
ConPaste & \hat{=} & \mathbf{if}\ dm_t = \mathsf{root}\ \mathbf{then}\ pt, pm_t := pu, pu \\
 & & \mathbf{else\ if}\ dm_t = \mathsf{left}\ \mathbf{then}\ (\sigma\ qm_t).left := pu;\ pm_t := pu \\
 & & \mathbf{else}\ (\sigma\ qm_t).right := pu;\ pm_t := pu\ \mathbf{fi}\ \mathbf{fi}.
\end{array}$$

Let DTV, V, MV and NT be as defined previously, and let DUV be defined as follows:

$$DUV \quad \hat{=} \quad Disjoint(u, pu, v, pv, \sigma).$$

The non-interference law for paste is:

Law 26 (Non-interference for Paste) Let $Paste \ \hat{=}\ AbsPaste \parallel ConPaste$.

$$\begin{array}{rcl}
\{R_{25} \wedge DTV \wedge DUV\}\ Paste\ [DTV \wedge DUV] & \leq & Paste \\
\{R_{25} \wedge DTV \wedge DUV \wedge V\}\ Paste\ [V] & \leq & Paste \\
\{R_{25} \wedge DTV \wedge DUV \wedge MV\}\ Paste\ [MV] & \leq & Paste \\
\{R_{25} \wedge NT \wedge m_t \not\preceq n_t\}\ Paste\ [NT] & \leq & Paste.
\end{array}$$

8 Example Refinements

8.1 Insert Operation

We apply some of the tree-refinement laws to the derivation of an implementation of the *Insert* operation shown in Figure 2. We wish to refine $\mathcal{A}[\![T,\,VT]\!](Insert)$, where $T \mathrel{\hat{=}} AbsTree(t, pt, \sigma)$, and $VT \mathrel{\hat{=}} (pt, \sigma)$. Since *Insert* introduces path m_t, we also define $MT \mathrel{\hat{=}} AbsTreePath(t, m_t, pt, pm_t, qm_t, dm_t, \sigma)$, and $VMT \mathrel{\hat{=}} (pt, pm_t, qm_t, dm_t, \sigma)$, then proceed as follows:

$$\begin{array}{ll}
 & \mathcal{A}[\![T,\,VT]\!](\\
 & |[\,\mathbf{var}\ m_t \cdot \quad m_t := \langle\rangle;\ \ \mathbf{do}\ \ldots\mathbf{od};\ \ \mathbf{if}\ \ldots\mathbf{fi}\ \ ]|) \\
\leq & \qquad \text{“ \textit{distribute} } \mathcal{A}\text{\textit{, extending abstraction relation and augmentation list} ”} \\
 & |[\,\mathbf{var}\ m_t, pm_t, qm_t, dm_t \cdot \\
 & \quad \mathcal{A}[\![MT,\,VMT]\!](m_t := \langle\rangle); \qquad (5) \\
 & \quad \mathcal{A}[\![MT,\,VMT]\!](\mathbf{do}\ \ldots\mathbf{od}); \qquad (6) \\
 & \quad \mathcal{A}[\![MT,\,VMT]\!](\mathbf{if}\ \ldots\mathbf{fi})\ \ ]| \qquad (7)
\end{array}$$

$$\begin{array}{ll}
(5)\ \leq & \qquad \text{“ \textit{refine initialise path} ”} \\
 & m_t := \langle\rangle\ \ ||\ \ pm_t, qm_t, dm_t := pt, pt, \mathsf{root}
\end{array}$$

$$\begin{array}{ll}
(6)\ \leq & \qquad \text{“ \textit{distribute} } \mathcal{A} \text{ \textit{through DO, transform guards} ”} \\
 & \mathbf{do}\ pm_t \neq \mathsf{nil}\ \wedge\ (\sigma\ pm_t).root \neq b \rightarrow \\
 & \qquad \mathbf{if}\ (\sigma\ pm_t).root \succ b \\
 & \qquad\ \ \mathbf{then}\ \mathcal{A}[\![MT,\,VMT]\!](\{Is_node(t/m_t)\}\ m_t := m_t\,\hat{}\,\langle\mathsf{left}\rangle) \qquad (8) \\
 & \qquad\ \ \mathbf{else}\ \mathcal{A}[\![MT,\,VMT]\!](\{Is_node(t/m_t)\}\ m_t := m_t\,\hat{}\,\langle\mathsf{right}\rangle)\ \mathbf{fi} \qquad (9) \\
 & \mathbf{od}.
\end{array}$$

$$\begin{array}{ll}
(8)\ \leq & \qquad \text{“ \textit{refine left path} ”} \\
 & m_t := m_t\,\hat{}\,\langle\mathsf{left}\rangle\ \ ||\ \ pm_t, qm_t, dm_t := (pm_t\ \sigma).left, pm_t, \mathsf{left}.
\end{array}$$

$$\begin{array}{ll}
(7)\ \leq & \qquad \text{“ \textit{distribute} } \mathcal{A} \text{ \textit{through IF, transform guard} ”} \\
 & \mathbf{if}\ pm_t = \mathsf{nil}\ \mathbf{then} \\
 & \qquad \mathcal{A}[\![MT,\,VMT]\!](t := t[m_t \backslash \mathsf{node}(b, \epsilon, \epsilon)]). \qquad (10)
\end{array}$$

Let $U \mathrel{\hat{=}} AbsTree(u, pu, \sigma)$, $DTU \mathrel{\hat{=}} Disjoint(t, pt, u, pu, \sigma)$.

$$\begin{array}{ll}
(10)\ \leq & \qquad \text{“\textit{introduce local variable}”} \\
 & \mathcal{A}[\![MT,\,VMT]\!](\\
 & |[\,\mathbf{var}\ u : Tree \cdot\ \ u := \mathsf{node}(b, \epsilon, \epsilon);\ \ t, u := t[m_t \backslash u], *\ ]|) \\
\leq & \qquad \text{“\textit{distribute} } \mathcal{A}\text{\textit{, extending abstraction relation and augmentation list} ”} \\
 & |[\,\mathbf{var}\ u, pu \cdot \\
 & \quad \mathcal{A}[\![MT \wedge U \wedge DTU, (VMT, pu)]\!](u := \mathsf{node}(b, \epsilon, \epsilon)) \qquad (11) \\
 & \quad \mathcal{A}[\![MT \wedge U \wedge DTU, (VMT, pu)]\!](t, u := t[m_t \backslash u], *)\ ]|\,. \qquad (12)
\end{array}$$

Continuing with (11), we deal with non-interference:

$$\begin{array}{ll}
 & (11) \\
\leq & \qquad \text{“\textit{assert abstraction relation}”}
\end{array}$$

$$
\begin{array}{l}
\mathbf{proc}\ \mathit{Insert}'(\mathbf{valres}\ pt : \mathit{TreePtr};\ \mathbf{valres}\ \sigma : \mathit{TreeStore};\ \mathbf{val}\ b : \mathit{Item})\ \hat{=} \\
|[\mathbf{var}\ pm_t, qm_t : \mathit{TreePtr};\ \ dm_t : \mathit{Dir}\cdot \\
pm_t, qm_t, dm_t := pt, pt, \mathsf{root}; \\
\mathbf{do}\ pm_t \neq \mathsf{nil}\ \wedge\ (\sigma\ pm_t).root \neq b \rightarrow \\
\quad \mathbf{if}\ (\sigma\ pm_t).root \succ b\ \mathbf{then}\ pm_t, qm_t, dm_t := (\sigma\ pm_t).\mathit{left}, pm_t, \mathsf{left} \\
\quad \mathbf{else}\ pm_t, qm_t, dm_t := (\sigma\ pm_t).\mathit{right}, pm_t, \mathsf{right}\ \mathbf{fi} \\
\mathbf{od}; \\
\mathbf{if}\ pm_t = \mathsf{nil}\ \mathbf{then} \\
\quad |[\mathbf{var}\ pu : \mathit{TreePtr}\cdot \\
\quad \mathit{new\_node}(pu, \sigma);\ \ (\sigma\ pu).root, (\sigma\ pu).\mathit{left}, (\sigma\ pu).\mathit{right} := b, \mathsf{nil}, \mathsf{nil} \\
\quad \mathbf{if}\ dm_t = \mathsf{root}\ \mathbf{then}\ pt := pu \\
\quad \mathbf{else\ if}\ dm_t = \mathsf{left}\ \mathbf{then}\ (\sigma\ qm_t).\mathit{left} := pu \\
\quad \mathbf{else}\ (\sigma\ qm_t).\mathit{right} := pu\ \ \mathbf{fi}\ \mathbf{fi}\ \ ]| \\
\mathbf{fi}\ \ ]|
\end{array}
$$

Figure 6: Refined Insert procedure

$$
\begin{array}{ll}
 & \{MT \wedge DTU\}\ \ \mathcal{A}[\![MT \wedge U \wedge DTU, (VMT, pu)]\!](u := \mathsf{node}(b, \epsilon, \epsilon)) \\
\leq & \quad \text{“\textit{coerce abstraction relation}”} \\
 & \{MT \wedge DTU\}\ \ \mathcal{A}[\![U, (VMT, pu)]\!](u := \mathsf{node}(b, \epsilon, \epsilon))\ \ [MT \wedge DTU] \\
\leq & \quad \text{“}\ pt, pm_t, qm_t, dm_t\ \mathbf{nfi}\ U\ \text{”} \\
 & \{MT \wedge DTU\}\ \ \mathcal{A}[\![U, (\sigma, pu)]\!](u := \mathsf{node}(b, \epsilon, \epsilon))\ \ [MT \wedge DTU] \\
\leq & \quad \text{“\textit{refine new node}”} \\
 & \{MT \wedge DTU\}\ \ (u := \mathsf{node}(b, \epsilon, \epsilon) \mid\mid \mathit{ConNewNode})\ \ [MT \wedge DTU] \\
\leq & \quad \text{“\textit{non-interference for new node}”} \\
 & (u := \mathsf{node}(b, \epsilon, \epsilon) \mid\mid \mathit{ConNewNode}).
\end{array}
$$

Similarly, we can show that

$$(12)\quad \leq\quad t, u := t[m_t \backslash u], *\ \ \mid\mid\ \ \mathit{ConPaste}.$$

Finally a diminution step is performed to remove the abstract variables from the code. Since all coercions involving abstract variables are eliminated in the final code fragments, all guards are independent of the abstract variables, and all assignments to concrete variables are independent of the abstract variables, we simply have to remove the assignments to abstract variables, resulting in the procedure shown in Figure 6. This procedure can be refined further by showing that the assertion

$$
\begin{array}{rcl}
dm_t = \mathsf{root} & \Leftrightarrow & qm_t = pm_t \\
dm_t = \mathsf{left} & \Leftrightarrow & qm_t \neq pm_t \wedge (\sigma\ qm_t).root \succ b \\
dm_t = \mathsf{right} & \Leftrightarrow & qm_t \neq pm_t \wedge (\sigma\ qm_t).root \prec b
\end{array}
$$

holds after the DO-loop; this allows dm_t to be eliminated.

8.2 Delete Operation

A procedure for deleting an item from a tree while maintaining the binary search order is shown in Figure 7. The first loop locates the item to be removed, if it exists, at the root of t/m_t. In the case that $\mathit{righttree}(t/m_t)$ is empty, the removal of the node pointed to by m_t is straightforward. In the case that $\mathit{righttree}(t/m_t)$ is

$$
\begin{array}{l}
\textbf{proc } \mathit{Delete}(\textbf{valres } t : \mathit{Tree};\ \textbf{val } b : \mathit{Item}) \mathrel{\hat{=}} \\
|[\textbf{var } m_t \cdot \quad m_t := \langle\rangle; \\
\textbf{do } t/m_t \neq \epsilon \ \wedge \ \mathit{roottree}(t/m_t) \neq b \rightarrow \\
\quad \textbf{if } \mathit{roottree}(t/m_t) \succ b \textbf{ then } m_t := m_t \hat{\ } \langle l \rangle \\
\quad \textbf{else } m_t := m_t \hat{\ } \langle r \rangle \textbf{ fi} \\
\textbf{od}; \\
\textbf{if } t/m_t = \epsilon \textbf{ then skip} \\
\textbf{else if } \mathit{righttree}(t/m_t) = \epsilon \\
\textbf{then } t := t[m_t \backslash \mathit{lefttree}(t/m_t)] \\
\textbf{else} \quad |[\textbf{var } n_t : \mathit{Path} \cdot \quad n_t := m_t \hat{\ } \langle r \rangle; \\
\qquad \textbf{do } \mathit{lefttree}(t/n_t) \neq \epsilon \rightarrow n_t := n_t \hat{\ } \langle l \rangle \textbf{ od}; \\
\qquad |[\textbf{var } u, v : \mathit{Tree} \cdot \quad t, u := t \upharpoonright n_t,\ t/n_t; \\
\qquad u, v := u \upharpoonright \langle \mathsf{right} \rangle,\ u/\langle \mathsf{right} \rangle; \\
\qquad t := t[n_t \backslash v]; \\
\qquad \{\mathit{Is\_node}(u)\} \\
\qquad t := t[m_t \backslash \mathsf{node}(\mathit{roottree}\ u, \mathit{lefttree}(t/m_t), \mathit{righttree}(t/m_t))] \quad]| \quad]| \\
\textbf{fi fi} \quad]|
\end{array}
$$

Figure 7: Delete procedure

not empty, the leftmost node (i.e., minimum item) of $\mathit{righttree}(t/m_t)$ is located and replaces the item to be removed; the tree is then rearranged accordingly.

The *Delete* procedure is not yet at a stage where the refinement laws can be applied directly so it needs to be refined in places. Rather than presenting a full refinement of the procedure here, we present a refinement of the final assignment:

$$t := t[m_t \backslash \mathsf{node}(\mathit{roottree}\ u, \mathit{lefttree}(t/m_t), \mathit{righttree}(t/m_t))].$$

Firstly, this needs to be broken into a number of smaller operations. Before proceeding we note the following program transformation: since a local variable is dropped when the block containing it is exited, the final statement of the block can always make an arbitrary assignment to the local variable, i.e., assume T does not assign to x, then

$$|[\textbf{var } x \cdot S;\ T]| \leq |[\textbf{var } x \cdot S;\ (T \parallel x := ?)\ ]|,$$

where $x := ? \quad = \quad x := x' \mid \mathit{true}$. Now

$$
\begin{array}{ll}
& \{\mathit{Is\_node}(t/m_t)\}\ t := t[m_t \backslash \mathsf{node}(\mathit{roottree}\ u, \mathit{lefttree}(t/m_t), \mathit{righttree}(t/m_t))] \\
\leq & \qquad \text{“ } t[m\backslash u] = (t \upharpoonright m)[m \backslash u] \text{ ”} \\
& \{\mathit{Is\_node}(t/m_t)\}\ t := (t \upharpoonright m_t)[m_t \backslash \mathsf{node}(\mathit{roottree}\ u, \mathit{lefttree}(t/m_t), \mathit{righttree}(t/m_t))] \\
\leq & \qquad \text{“ } \mathit{introduce}\ w,\ \mathit{cut}\ t \text{ ”} \\
& |[\textbf{var } w : \mathit{Tree} \cdot \ t, w := t \upharpoonright m_t,\ t/m_t; \\
& \{\mathit{Is\_node}(w)\}\ t, w := t[m_t \backslash \mathsf{node}(\mathit{roottree}\ u, \mathit{lefttree}\ w, \mathit{righttree}\ w)], ?\ ]| \qquad (13)
\end{array}
$$

$$
\begin{array}{lll}
(13) & \leq & \qquad \text{“}\mathit{introduce}\ l\ \mathit{and}\ r\text{ ”} \\
& & |[\textbf{var } l, r : \mathit{Tree} \cdot \ \{\mathit{Is\_node}(w)\}\ w, l, r := w \upharpoonright \langle \mathsf{left} \rangle \upharpoonright \langle \mathsf{right} \rangle,\ w/\langle \mathsf{left} \rangle,\ w/\langle \mathsf{right} \rangle; \\
& & t, l, r := t[m_t \backslash \mathsf{node}(\mathit{roottree}\ u, l, r)], *, *\ ]| \qquad (14)
\end{array}
$$

$$
\begin{array}{lll}
(14) & \leq & \text{“ \textit{introduce arbitrary assignment} ”} \\
& & \{Is\_node(u)\}\ t, l, r, u := t[m_t \backslash \mathsf{node}(roottree\ u, l, r)], *, *, ? \\
& \leq & \text{“\textit{leading assignment} ”} \\
& & \{Is\_node(u)\}\ u, l, r := u[\langle \mathsf{left} \rangle \backslash l][\langle \mathsf{right} \rangle \backslash r], *, *;\quad t, u := t[m_t \backslash u], *.
\end{array}
$$

Augmentation and diminution can now be applied, resulting in

$$
\begin{array}{l}
|[\mathbf{var}\ pw, pl, pr : TreePtr \cdot \\
pw := pm_t; \\
pl, pr := (\sigma\ pw).left,\ (\sigma\ pw).right; \\
(\sigma\ pu).left,\ (\sigma\ pu).right := pl, pr; \\
ConPaste\ ]|\ .
\end{array}
$$

Using laws of the refinement calculus, this can be simplified to:

$$(\sigma\ pu).left,\ (\sigma\ pu).right := (\sigma\ pm_t).left, (\sigma\ pm_t).right; \quad ConPaste.$$

9 Guidelines

Before the laws can be applied to abstract specifications of programs, the specifications must be refined so that they are described in terms of the abstract operations to which the laws apply, e.g., operations for moving to the left subtree, moving to the right subtree, cutting a tree, and pasting trees together. This refinement will involve properties of trees such as the following, which were used in the derivation of the *Member*, *Insert* and *Delete* procedures:

$$
\begin{array}{l}
m \in paths\ t \ \Rightarrow \\
\qquad \begin{array}{rcl}
elems(t \restriction m) & = & elems(t) \setminus elems(t/m) \\
elems(t[m \backslash u]) & = & elems(t \restriction m) \cup elems(u).
\end{array}
\end{array}
$$

Such intuitive laws are easily proved using structural induction on trees. Note that it is intended that the $*$-tree should only appear during intermediate stages of tree manipulation, and not in the initial or final values.

An abstract tree t should be represented by the pair pt, σ using the abstraction invariant $AbsTree(t, pt, \sigma)$. If there is more than one tree within scope, then the representations should share σ, and for each pair of distinct trees t, u, $Disjoint(t, pt, u, pu, \sigma)$ should be included in the abstraction invariant. Each path m_t should be represented by the variables $pt, pm_t, qm_t, dm_t, \sigma$, and $AbsTreePath(t, m_t, pt, pm_t, qm_t, dm_t, \sigma)$ should be included in the abstraction invariant. It is best to keep local tree and path variables as localised as possible to avoid too much non-interference checking. In the case that non-interference checking is not required, or can be trivially discharged, then straight data-refinement is sufficient (e.g., the derivation of $Member'$), otherwise it is necessary to split the data refinement into an augmentation step and a diminution step so that abstraction relations can be introduced explicitly as assertions and coercions.

The set of abstract operations for which we provide refinement rules was chosen by informal analysis of procedures on trees such as *Member*, *Insert* and *Delete*, and we make no claim about their completeness, i.e., we do not guarantee that all pointer algorithms involving tree-based structures can be derived using them. Clearly other abstract operations having simple pointer implementations could be identified leading to new refinement laws. However, often these new laws can be justified in terms of the laws we have provided rather than having to appeal to the general data-refinement laws. Several such derived laws are presented in the appendix. For example, the *Paste Left* law is verified by observing that

$$
\begin{array}{l}
\{t \neq \epsilon\};\ t, u := t[\langle \mathsf{left} \rangle \backslash u], * \ = \\
\quad |[\mathbf{var}\ m_t \cdot m_t := \langle\rangle;\ \{t \neq \epsilon\};\ m_t := m_t \frown \langle \mathsf{left} \rangle;\ t, u := t[m_t \backslash u], *]|\ .
\end{array}
$$

The refinement laws for trees are applicable directly to the right-hand side, and the resulting code can then be optimised to $(\sigma\ pt).\mathit{left} := pu$.

10 Conclusions

Several general approaches to dealing with the correctness of pointer algorithms exist [3, 7, 8]. These are mostly suited to program verification rather than the more calculational program-refinement style, and their use can be quite cumbersome as they usually involve complex assertions. The approach of Möller [9] is more suited to the calculational style, though we still found the derivations of algorithms involving tree-based structures to be quite involved.

We found that the difficulty this caused could be alleviated by identifying some commonly used pointer manipulations on tree-based structures and providing representations of these manipulations in terms of abstract trees. This allows most of the algorithmic refinement to be performed on abstract trees, which have cleaner properties, before introducing pointers. We required two simple enrichments of abstract trees: paths, which allow for access of subtrees without using destructive descent, and the $*$-tree, which provides a neat way of dealing with a certain form of sharing that arises during manipulation of structures. The laws for pushing data refinement through program structures also make it possible to introduce the algorithmic structure before introducing pointers. The general laws we introduced for decomposition of data refinement (Laws 11 to 13) make it possible to modularise the application of our refinement laws for trees.

Although we have concentrated on trees, it should be possible to apply the general approach to other abstract structures. This would involve identifying commonly used operations at the pointer level and finding appropriate abstract representations of those operations.

A Derived Laws

Law 27 (Refinement of New Node2)

$$\begin{array}{rcl}
a_{27} & \hat{=} & t_1, t_2, t_3 \\
c_{27} & \hat{=} & pt_1, pt_2, pt_3, \sigma \\
R_{27} & \hat{=} & \forall\, i : 1..3 \cdot \mathit{AbsTree}(t_i, pt_i, \sigma) \wedge \\
 & & \forall\, i, j : 1..3 \mid i \neq j \cdot \mathit{Disjoint}(t_i, pt_i, t_j, pt_j, \sigma) \\
\mathit{AbsNewNode2} & \hat{=} & t_1, t_2, t_3 := \mathsf{node}(b, t_2, t_3), *, * \\
\mathit{ConNewNode2} & \hat{=} & \mathit{new_node}(pt1);\ (\sigma\ pt).\mathit{root}, (\sigma\ pt).\mathit{left}, (\sigma\ pt).\mathit{right} := b, t_2, t_3.
\end{array}$$

Law 28 (Non-interference for New Node2) Let $D \mathrel{\hat{=}} \forall\, i : 1..3 \cdot \mathit{Disjoint}(t_i, pt_i, v, pv, \sigma)$, $\mathit{NewNode2} \mathrel{\hat{=}} \mathit{AbsNewNode2} \parallel \mathit{ConNewNode2}$.

$$\begin{array}{rcl}
\{R_{27} \wedge D\}\ \mathit{NewNode2}\ [D] & \leq & \mathit{NewNode2} \\
\{R_{27} \wedge D \wedge V\}\ \mathit{NewNode2}\ [V] & \leq & \mathit{NewNode2} \\
\{R_{27} \wedge D \wedge MV\}\ \mathit{NewNode2}\ [MV] & \leq & \mathit{NewNode2}.
\end{array}$$

Law 29 (Refinement of Paste Left)

$$\begin{array}{rcl}
a_{29} & \hat{=} & t, u \\
c_{29} & \hat{=} & pt, pu, \sigma \\
R_{29} & \hat{=} & \mathit{AbsTree}(t, pt, \sigma) \wedge \mathit{AbsTree}(u, pu, \sigma) \wedge \\
 & & \mathit{Disjoint}(t, pt, u, pu, \sigma) \\
\mathit{AbsPasteLeft} & \hat{=} & \{\mathit{Is_node}(t)\}\ t, u := t[\langle \mathsf{left} \rangle \backslash u], * \\
\mathit{ConPasteLeft} & \hat{=} & (\sigma\ pt).\mathit{left} := pu.
\end{array}$$

Law 30 (Non-interference for Paste Left) Let $PasteLeft \;\hat{=}\; AbsPasteLeft \,\|\, ConPasteLeft$.

$$\begin{array}{rcl} \{R_{29} \wedge DTV \wedge DUV\}\ PasteLeft\ [DTV \wedge DUV] & \leq & PasteLeft \\ \{R_{29} \wedge DTV \wedge DUV \wedge V\}\ PasteLeft\ [V] & \leq & PasteLeft \\ \{R_{29} \wedge DTV \wedge DUV \wedge MV\}\ PasteLeft\ [MV] & \leq & PasteLeft \\ \{R_{29} \wedge NT \wedge \langle \mathsf{left} \rangle \not\preceq n_t\}\ PasteLeft\ [NT] & \leq & PasteLeft. \end{array}$$

Law 31 (Refinement of Cut Left)

$$\begin{array}{rcl} a_{31} & \hat{=} & t, u \\ c_{31} & \hat{=} & pt, pu, \sigma \\ R_{31} & \hat{=} & AbsTree(t, pt, \sigma) \wedge AbsTree(u, pu, \sigma) \wedge \\ & & Disjoint(t, pt, u, pu, \sigma) \\ AbsCutLeft & \hat{=} & \{Is\_node(t)\}\ t, u := t \upharpoonright \langle \mathsf{left} \rangle, t / \langle \mathsf{left} \rangle \\ ConCutLeft & \hat{=} & pu := (\sigma\ pt).left. \end{array}$$

Law 32 (Non-interference for Cut Left) Let $CutLeft \;\hat{=}\; AbsCutLeft \,\|\, ConCutLeft$.

$$\{R_{31} \wedge NT \wedge \langle \mathsf{left} \rangle \not\prec n_t\}\ CutLeft\ [NT] \;\; \leq \;\; CutLeft.$$

Paste right and cut right are similarly defined.

Law 33 (Refinement of Cut Left-Right)

$$\begin{array}{rcl} a_{33} & \hat{=} & t_1, t_2, t_3 \\ c_{33} & \hat{=} & pt_1, pt_2, pt_3, \sigma \\ R_{33} & \hat{=} & \forall\, i : 1..3 \cdot AbsTree(t_i, pt_i, \sigma) \wedge \\ & & \forall\, i, j : 1..3 \mid i \neq j \cdot Disjoint(t_i, pt_i, t_j, pt_j, \sigma) \\ AbsCutLeftRight & \hat{=} & \{Is\_node(t_1)\}\ t_1, t_2, t_3 := t_1 \upharpoonright \langle \mathsf{left} \rangle \upharpoonright \langle \mathsf{right} \rangle, t_1 / \langle \mathsf{left} \rangle, t_1 / \langle \mathsf{right} \rangle \\ ConCutLeftRight & \hat{=} & pt_2, pt_3 := (\sigma\ pt_1).left, (\sigma\ pt_1).right. \end{array}$$

Law 34 (Non-interference for Cut Left-Right) Let $CutLeftRight \;\hat{=}\; AbsCutLeftRight \,\|\, ConCutLeftRight$.

$$\begin{array}{rcl} \{R_{33} \wedge D\}\ CutLeftRight\ [D] & \leq & CutLeftRight \\ \{R_{33} \wedge D \wedge V\}\ CutLeftRight\ [V] & \leq & CutLeftRight \\ \{R_{33} \wedge D \wedge MV\}\ CutLeftRight\ [MV] & \leq & CutLeftRight. \end{array}$$

References

[1] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[2] R.J.R. Back and M.J. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction, 1995*, volume LNCS 947. Springer–Verlag, 1995.

[3] A. Bijlsma. Calculating with pointers. *Science of Computer Programming*, 12:191–205, 1989.

[4] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[5] P.H.B. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[6] C.A.R. Hoare, He Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.

[7] D.C. Luckham and N. Suzuki. Verification of array, record, and pointer operations in pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, 1979.

[8] I. Mason. Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10:177–210, 1988.

[9] B. Möller. Towards pointer algebra. *Science of Computer Programming*, 21:57–90, 1993.

[10] C.C. Morgan. *Programming from Specifications, Second Edition.* Prentice–Hall, 1994.

[11] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Prog.*, 9(3):298–306, 1987.

[12] J. von Wright. The lattice of data refinement. *Acta Informatica*, 31(2):105–135, 1994.

[13] J. von Wright. *Program Refinement by Theorem Prover.* In *6th Refinement Workshop, London.* Springer–Verlag, 1994.