Calculational Derivation of Pointer Algorithms from Tree Operations

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

June 12, 1998

Abstract

We describe an approach to the derivation of correct algorithms on tree-based pointer structures. The approach is based on enriching trees in a way that allows us to model commonly-used pointer manipulations on tree structures. We provide rules which allow recursive functions on trees to be transformed into imperative algorithms on enriched trees. In addition, we provide rules which allow algorithms on enriched trees to be mechanically transformed into efficient pointer algorithms. All transformations are correctness-preserving. A key point of our approach is that we avoid aliasing through the way in which trees are enriched and through some simple syntactic restrictions on transformable programs.

Keywords: Pointers; Trees; Correctness; Refinement; Transformation.

1 Introduction

The derivation of correct algorithms involving pointers is difficult. In particular, the potentially complex connections within pointer structures make it difficult to reason about the side effects of pointer assignment. This is further complicated by having to reason about algorithmic structures including loops and recursion.

In this paper, we describe an approach that factors out and simplifies some of these difficulties in the context of tree-based pointer structures. We view the derivation of algorithms on tree-based pointer structures as an instance of data refinement, that is, the replacement of abstract trees by concrete pointer structures. Operations on abstract trees are more naturally described as applicative-style recursive functions, whereas operations on pointer structures are more naturally described as imperative algorithms. We factor out the difficulties of the derivation of pointer implementations by enriching abstract trees in a way that allows algorithms on them to be described in an imperative fashion. Thus we can develop the necessary algorithmic structure before introducing the pointer data structures. We further simplify the derivation by providing a collection of pre-proved specialised refinement transformations.

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

$$
\begin{array}{lcl}
member(a, \epsilon) & = & false \\
member(a, \mathsf{node}(b, L, R)) & = & \mathbf{if}\ a = b \rightarrow true \\
 & & [\!]\ \ a < b \rightarrow member(a, L) \\
 & & [\!]\ \ a > b \rightarrow member(a, R)\ \mathbf{fi}.
\end{array}
$$

Figure 1: Applicative search function on binary trees

paths are represented at the concrete level by pointers. The $*$-tree provides an elegant way of dealing with sharing that arises in intermediate stages of pointer structure manipulation. The $*$-tree allows us to avoid aliasing of pointer structures, thus removing many of the difficulties that usually arise when reasoning about pointer structures. We describe some operations on enriched trees and provide rules for transforming these into simple standard pointer operations. These rules are compositional and allow a correct pointer implementation to be mechanically calculated from an algorithm involving enriched trees.

This still leaves a gap between the standard applicative-style descriptions of tree operations and imperative algorithms on enriched trees. To help bridge this gap, we provide a collection of rules that allow imperative algorithms on enriched trees to be calculated from standard descriptions of tree operations. In some cases, as will be made clear, the resulting programs require further refinement before the pointer-introduction transformations may be applied, which means that the full derivation path is not completely mechanical.

The program derivation framework used in our approach is the *refinement calculus* of Back, Morgan and Morris [2, 14, 16]. The refinement calculus is a formalisation of the stepwise refinement method of program construction. The required behaviour is specified as an abstract, possibly non-executable, program which is then refined by a series of correctness-preserving refinement transformations into an efficient, executable program. The programming notation used is Dijkstra's guarded command language [8], extended with *assertion* statements $\{p\}$ (assume predicate p holds at this point in the computation) and *generalised assignments* $x := x'|p$ (assign to x a value x' satisfying predicate p).

In order to describe the scope of this paper more clearly, we introduce some definitions associated with trees. Assume that *Item* is some given type, then a (binary) tree of *Item* is defined as a recursive data type:

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

That is, a tree can be the empty tree or a node consisting of an item and a left and a right subtree.

Typically, functions on trees are described as applicative-style recursive functions. For example, the function *member*, which checks whether an item exists in an ordered tree, is defined in Figure 1. The components of a node tree are accessed as follows:

$$
\begin{array}{rcl}
root(\mathsf{node}(b, L, R)) & \hat{=} & b \\
left(\mathsf{node}(b, L, R)) & \hat{=} & L \\
right(\mathsf{node}(b, L, R)) & \hat{=} & R.
\end{array}
$$

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

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

We write $\langle\rangle$ for the empty path, $\langle d \rangle$ for the singleton path and $m_1 m_2$ for the concatenation of paths m_1 and m_2. The subtree of t reached by following path m is

$$
\begin{array}{l}
\textbf{procedure } \mathit{Member}(\textbf{val } t : \mathit{Tree};\ \textbf{val } a : \mathit{Item};\ \textbf{ref } x : \mathit{Bool}) \mathrel{\hat{=}} \\
\quad |[\textbf{var } m : \mathit{Path} \bullet m := \langle\rangle; \\
\quad \textbf{do } t/m \neq \epsilon \ \wedge\ a \neq \mathit{root}(t/m) \rightarrow \\
\qquad \textbf{if } a < \mathit{root}(t/m) \rightarrow m := m\langle \mathsf{left} \rangle \\
\qquad [] \ \ a > \mathit{root}(t/m) \rightarrow m := m\langle \mathsf{right} \rangle \textbf{ fi} \\
\quad \textbf{od}; \\
\quad x := (t/m \neq \epsilon) \\
\quad]|.
\end{array}
$$

Figure 2: Imperative search procedure on binary trees

denoted t/m which is (partially) defined as follows:

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

The set of all paths of a tree is defined as follows:

$$
\begin{array}{lcl}
\mathit{paths}(\epsilon) & \hat{=} & \{ \ \langle\rangle \ \} \\
\mathit{paths}(\mathsf{node}(b, L, R)) & \hat{=} & \{ \ \langle\rangle \ \} \ \cup \ \{ \ \langle \mathsf{left} \rangle m \mid m \in \mathit{paths}(L) \ \} \\
 & & \qquad\quad \cup \ \{ \ \langle \mathsf{right} \rangle m \mid m \in \mathit{paths}(R) \ \}.
\end{array}
$$

The *Member* procedure of Figure 2 is a refinement of the statement $x := \mathit{member}(a, t)$[1]. Here, the path m is continually extended in the appropriate direction until a node is reached that contains the item a or until the empty tree is reached.

Given trees t_i, t_j, and path $m_i \in \mathit{paths}(t_i)$, $t_i[m_i \backslash t_j]$ represents the tree t_i with the subtree reached using m_i replaced by t_j:

$$
\begin{array}{rcl}
t_i \ [\ \langle\rangle \ \backslash \ t_j \ ] & \hat{=} & t_j \\
\mathsf{node}(b, L, R) \ [\ \langle \mathsf{left} \rangle m_i \ \backslash \ t_j \ ] & \hat{=} & \mathsf{node}(b, \ L[m_i \backslash t_j], \ R) \\
\mathsf{node}(b, L, R) \ [\ \langle \mathsf{right} \rangle m_i \ \backslash \ t_j \ ] & \hat{=} & \mathsf{node}(b, \ L, \ R[m_i \backslash t_j]).
\end{array}
$$

For tree t and item b, $t \rhd b$ represents t with the item at the root replaced by b, and $t[m \rhd b]$ represents t with the item at the root of the subtree reached by path m replaced by b:

$$
\begin{array}{rcl}
\mathsf{node}(a, L, R) \rhd b & \hat{=} & \mathsf{node}(b, L, R) \\
t[m \rhd b] & \hat{=} & t[m \backslash (t/m \rhd b)].
\end{array}
$$

The use of paths to access and manipulate subtrees is common in term-rewriting literature [7].

Scope and Outline

In this paper, we present pointer-introduction rules that transform an imperative algorithm on enriched trees into an algorithm on pointer structures. For example, the procedure in Figure 2 is transformed into the pointer procedure in Figure 3. These transformations are presented in two stages: Section 3 describes transformations on operations that do not manipulate the structure of a tree, while Section 6 describes transformations on operations that do manipulate the structure of a tree. These pointer-introduction transformations may be fully mechanised. The proof of soundness of these transformations is outlined in Section 8.

[1]The keyword **val** indicates call-by-value parameters, while **ref** indicates call-by-reference.

$$
\begin{array}{l}
\textbf{procedure } \mathit{Member}(\textbf{val } p : \mathit{TreePtr};\ \textbf{val } a : \mathit{Item};\ \textbf{ref } x : \mathit{Bool}) \mathrel{\hat{=}} \\
\quad |[\textbf{var } q : \mathit{TreePtr} \bullet q := p; \\
\quad \textbf{do } q \neq \mathsf{nil} \ \wedge \ a \neq q\hat{}.\mathit{root} \rightarrow \\
\qquad \textbf{if } a < q\hat{}.\mathit{root} \rightarrow q := q\hat{}.\mathit{left} \\
\qquad [] \ \ a > q\hat{}.\mathit{root} \rightarrow q := q\hat{}.\mathit{right} \ \textbf{fi} \\
\quad \textbf{od}; \\
\quad x := (q \neq \mathsf{nil}) \\
\quad]|.
\end{array}
$$

Figure 3: Search procedure on pointer structures

In addition, we present some rules that may be used to calculate algorithms on enriched trees, such as Figure 2, from applicative-style recursive functions, such as Figure 1. These rules are presented in two sections, Sections 4 and 7, to mirror the presentation of the pointer-introduction transformations.

Some of the ideas presented in this paper appear in an earlier paper [4]. However, that paper did not present a full pointer-introduction transformation mechanism, rather a set of rules that could be used to assist the transformation. These rules were many and complicated, making them difficult to remember and use. We have overcome this difficulty by providing a safe syntactic transformation function that could easily be mechanised. Furthermore, that earlier paper did not deal with the transformation of applicative-style recursive functions on trees into algorithms on enriched trees.

We do not claim to provide an approach for dealing with any pointer algorithm. Rather our approach is most suited to algorithms that are naturally described in terms of recursive data structures such as lists and trees.

We proceed with a general consideration of pointer structures.

2 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;
```

A structure of this type may be modelled explicitly as a partial map (i.e., a many-to-one set of pairs) from pointers to records:

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

Here we assume that *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 p be a pointer to a record in this store, i.e., $p \in \mathit{dom}(\sigma)$. Then a reference to $p\hat{}$ represents the record $\sigma(p)$ and a reference to $p\hat{}.\mathit{root}$ represents the item $\sigma(p).\mathit{root}$.

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

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

In conformance with standard programming notation, no explicit references to the store σ will be made in code fragments. Instead, occurrence of the dereferencing symbol (^) will imply a reference to the store.

We assume that the value nil is of type *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 (with respect to σ) if it is in the domain of σ or if it equals nil.

The **new**(p) statement that allocates space for a new record in the store is modelled by the following nondeterministic assignment that selects an unused pointer and creates a new record with arbitrary values:

$$\begin{array}{ll} p, \sigma := p', \sigma' \ | & p' \notin dom(\sigma) \ \wedge \ p' \neq \mathsf{nil} \ \wedge \\ & \exists b, pl, pr \cdot \sigma' = \sigma \cup \{(p', \ \ mk\_node(b, pl, pr))\}. \end{array}$$

The **dispose**(p) statement that deallocates the space used by the record pointed to by p is modelled by an operation that removes all entries for p from the store σ:

$$\sigma \ := \ \sigma \ \setminus \ (\{p\} \times Node).$$

A function that extracts the tree modelled by a pointer structure is defined as:

$$\begin{array}{l} get\_tree(\sigma, p) \ = \\ \quad \textbf{if} \ p = \mathsf{nil} \rightarrow \epsilon \\ \quad [] \ \ p \neq \mathsf{nil} \rightarrow \mathsf{node}(\ \sigma(p).root, \ \ get\_tree(\sigma, \sigma(p).left), \ \ get\_tree(\sigma, \sigma(p).right) \ ) \\ \quad \textbf{fi}. \end{array}$$

We assume a strict semantics for our language of applicative functions, thus $get\_tree(\sigma, p)$ will only give a well-defined result if p and each of its descendants is well defined in σ and the structure in σ starting at p is acyclic.

3 Pointer-Introduction Transformation I

In this section, we present a set of rules for transforming certain specifications involving trees and paths into specifications involving pointer structures. The transformations do not deal with manipulation of these structures, but will be extended in Section 6 to deal with this.

The pointer-introduction transformations replace each abstract tree t_i by a pointer p_i and a single store σ which will be shared by all trees t_i, t_j, etc. Having a single shared store allows us to model the swapping of nodes between trees. However, the pointer-introduction rules that we present are such that only a limited and safe form of sharing is allowed. This is described in Section 5.

A tree t_i can have at most one path m_i associated with it. Rather than representing a path m_i directly at the concrete level, we instead represent the subtree t_i/m_i with a pointer q_i.

The result of the pointer-introduction transformations is guaranteed to be a correct data refinement of the source specification preserving the following abstraction relation between trees, paths, pointers and the store:

$$\begin{array}{rcl} get\_tree(\sigma, p_i) & = & t_i \\ get\_tree(\sigma, q_i) & = & t_i/m_i. \end{array}$$

The largest unit of specification is a module which consists of program variables, procedures and a main body (S):

module *Mod*
var t_I, m_J, x
procedure $Proc_1(\ t_{I_1}, m_{J_1}, x_1\ )\ \ \hat{=}\ \ S_1$
$\vdots$
procedure $Proc_N(\ t_{I_N}, m_{J_N}, x_N\ )\ \ \hat{=}\ \ S_N$
begin
S
end

Here t_I represents a set of tree variables indexed by I (e.g., $t_1, \ldots, t_n$). We assume that a module has tree variables t_I, path variables m_J and other variables represented by x. Each path m_i is associated with tree t_i and $J \subseteq I$. Each procedure $Proc_i$ may have tree parameters t_{I_i}, path parameters m_{J_i} ($J_i \subseteq I_i$) and other parameters x_i.

A module of the above form is transformed to a module of the following form:

module *Mod*
var p_I, q_J, x
procedure $Proc_1(\ p_{I_1}, q_{J_1}, x_1\ )\ \ \hat{=}\ \ \mathcal{S}[\![\ S_1\ ]\!]$
$\vdots$
procedure $Proc_N(\ p_{I_N}, q_{J_N}, x_N\ )\ \ \hat{=}\ \ \mathcal{S}[\![\ S_N\ ]\!]$
begin
$\mathcal{S}[\![\ S\ ]\!]$
end

Each tree t_i is replaced by a pointer p_i and each path m_i is replaced by a pointer q_i. Other variables (x) are unchanged. Similar changes are made to the parameters of each procedure. The call mode of each parameter (i.e., **val** or **ref**) is the same as the corresponding abstract parameter.

The transformation $\mathcal{S}$ is defined on the structure of statements. Transformation of compound statements distributes through the structure:

$$
\begin{array}{lcl}
\mathcal{S}[\![\ S_1;\ S_2\ ]\!] & \hat{=} & \mathcal{S}[\![\ S_1\ ]\!];\ \mathcal{S}[\![\ S_2\ ]\!] \\
\mathcal{S}[\![\ \{\ P\ \}\ ]\!] & \hat{=} & \{\ \mathcal{E}[\![\ P\ ]\!]\ \} \\
\mathcal{S}[\![\ \mathbf{do}\ P \rightarrow S\ \mathbf{od}\ ]\!] & \hat{=} & \mathbf{do}\ \mathcal{E}[\![\ P\ ]\!] \rightarrow \mathcal{S}[\![\ S\ ]\!]\ \mathbf{od} \\
\multicolumn{3}{l}{\mathcal{S}[\![\ \mathbf{if}\ P_1 \rightarrow S_1\ [\!]\ \cdots\ [\!]\ P_n \rightarrow S_n\ \mathbf{fi}\ ]\!]} \\
 & \hat{=} & \mathbf{if}\ \mathcal{E}[\![\ P_1\ ]\!] \rightarrow \mathcal{S}[\![\ S_1\ ]\!]\ [\!]\ \cdots\ [\!]\ \mathcal{E}[\![\ P_n\ ]\!] \rightarrow \mathcal{S}[\![\ S_n\ ]\!]\ \mathbf{fi}.
\end{array}
$$

The transformation $\mathcal{E}$ on expressions (including predicates) is defined below.

An assignment to program variables that do not include any of t_I or m_J is transformed by applying transformation $\mathcal{E}$ to each expression on the right hand side:

$$\mathcal{S}[\![\ x_1, \ldots, x_n := E_1, \ldots, E_n\ ]\!]\ \ \hat{=}\ \ x_1, \ldots, x_n := \mathcal{E}[\![\ E_1\ ]\!], \ldots, \mathcal{E}[\![\ E_n\ ]\!].$$

The transformation $\mathcal{E}$ distributes through the structure of an expression leaving it unchanged except that references, of the following form, to the root of a tree are

$root(t_i)$	$t_i = \epsilon$
$root(left(t_i))$	$left(t_i) = \epsilon$
$root(right(t_i))$	$right(t_i) = \epsilon$
$root(t_i/m_i)$	$t_i/m_i = \epsilon$
$root(left(t_i/m_i))$	$left(t_i/m_i) = \epsilon$
$root(right(t_i/m_i))$	$right(t_i/m_i) = \epsilon$

Figure 4: Transformable tree expressions.

replaced by pointer expressions:

$$\begin{array}{lcl}
\mathcal{E}[\![\ root(t_i)\ ]\!] & \hat{=} & p_i\hat{}.root \\
\mathcal{E}[\![\ root(left(t_i))\ ]\!] & \hat{=} & (p_i\hat{}.left)\hat{}.root \\
\mathcal{E}[\![\ root(right(t_i))\ ]\!] & \hat{=} & (p_i\hat{}.right)\hat{}.root \\
\mathcal{E}[\![\ root(t_i/m_i)\ ]\!] & \hat{=} & q_i\hat{}.root \\
\mathcal{E}[\![\ root(left(t_i/m_i))\ ]\!] & \hat{=} & (q_i\hat{}.left)\hat{}.root \\
\mathcal{E}[\![\ root(right(t_i/m_i))\ ]\!] & \hat{=} & (q_i\hat{}.right)\hat{}.root.
\end{array}$$

Furthermore, checks that a tree, of the following form, is empty are replaced by checks that a pointer is nil:

$$\begin{array}{lcl}
\mathcal{E}[\![\ t_i = \epsilon\ ]\!] & \hat{=} & p_i = \mathsf{nil} \\
\mathcal{E}[\![\ left(t_i) = \epsilon\ ]\!] & \hat{=} & p_i\hat{}.left = \mathsf{nil} \\
\mathcal{E}[\![\ right(t_i) = \epsilon\ ]\!] & \hat{=} & p_i\hat{}.right = \mathsf{nil} \\
\mathcal{E}[\![\ t_i/m_i = \epsilon\ ]\!] & \hat{=} & q_i = \mathsf{nil} \\
\mathcal{E}[\![\ left(t_i/m_i) = \epsilon\ ]\!] & \hat{=} & q_i\hat{}.left = \mathsf{nil} \\
\mathcal{E}[\![\ right(t_i/m_i) = \epsilon\ ]\!] & \hat{=} & q_i\hat{}.right = \mathsf{nil}.
\end{array}$$

For example, assume Q contains no references to any of t_I or m_J, then

$$\mathcal{E}[\![\ \neg(t_i/m_i = \epsilon)\ \wedge\ root(t_i/m_i) \geq a\ \wedge\ Q\ ]\!] \quad = \quad \neg(q_i = \mathsf{nil})\ \wedge\ q_i\hat{}.root \geq a\ \wedge\ Q.$$

The set of transformable subexpressions involving trees is reiterated in Figure 4. References to trees t_I or paths m_J that are not of these forms should not appear in E when constructing $\mathcal{E}[\![\ E\ ]\!]$. Thus, for example, while the expression $root(t_1) = root(t_2)$ is transformable, the expression $t_1 = t_2$ is not; the reason for this is that $t_1 = t_2$ cannot be implemented by a simple test at the pointer level.

A transformation such as $\mathcal{E}[\![\ root(t_i)\ ]\!]$ is valid even if $root(t_i)$ is undefined, i.e., when $t_i = \epsilon$. This is because an assignment such as $x := root(t_i)$ aborts when $root(t_i)$ is undefined and an aborting command may be replaced by any command, including $x := p_i\hat{}.root$. Such undefinedness is dealt with in Section 8. We make no assumption about the order of evaluation of conjunctions, so that, $t_i \neq \epsilon \wedge root(t_i) > a$ is equal to $root(t_i) > a\ \wedge\ t_i \neq \epsilon$.

3.1 Assignments to Trees and Paths

We do not provide transformations of arbitrary assignments to trees and paths, rather we provide transformations of assignments that have simple implementations as pointer operations. The first set of assignments to trees and paths for which direct transformations are provided is given in Figure 5.

$$
\begin{array}{l}
t_i := \epsilon \\
t_i := \mathsf{node}(b, \epsilon, \epsilon) \\
m_i := \langle\rangle \\
\{Is\_Node(t_i/m_i)\} \quad m_i := m_i\langle \mathsf{left} \rangle \\
\{Is\_Node(t_i/m_i)\} \quad m_i := m_i\langle \mathsf{right} \rangle \\
t_i := t_i \rhd E \\
t_i := t_i[m_i \rhd E]
\end{array}
$$

Figure 5: Transformable tree assignments.

Constructors

Construction of an empty tree is transformed to a simple pointer assignment:

$$\mathcal{S}[\![\ t_i := \epsilon\ ]\!] \;\hat{=}\; p_i := \mathsf{nil}.$$

Construction of a new node with empty subtrees is transformed to a call to **new** followed by appropriate assignments to the fields of the new record: provided E is independent of t_i, then

$$\begin{array}{l}\mathcal{S}[\![\ t_i := \mathsf{node}(E, \epsilon, \epsilon)\ ]\!] \;\hat{=} \\ \qquad\qquad \mathbf{new}(p_i);\;\; p_i\hat{}.root,\ p_i\hat{}.left,\ p_i\hat{}.right := \mathcal{E}[\![\ E\ ]\!],\ \mathsf{nil},\ \mathsf{nil}.\end{array}$$

For both these operations, it must be the case that t_i does not have an associated path variable m_i in the current scope.

Path Assignments

An operation for initialising the path m_i is transformed to an assignment to pointer q_i:

$$\mathcal{S}[\![\ m_i := \langle\rangle\ ]\!] \;\hat{=}\; q_i := p_i.$$

Let $Is\_Node(t) \;\hat{=}\; (\exists a, L, R \cdot t = \mathsf{node}(a, L, R))$[2]. (We use the convention throughout that boolean functions begin with an upper case letter.) An operation for performing a left extension of a path is transformed to an assignment that updates pointer q_i:

$$\mathcal{S}[\![\ \{Is\_Node(t_i/m_i)\}\ m_i := m_i\langle \mathsf{left}\rangle\ ]\!] \;\hat{=}\; q_i := q_i\hat{}.left.$$

Note the precondition on the path operation which ensures that the extended path is still a path of t_i. This is required because $m_i\langle \mathsf{left}\rangle$ is always well defined on its own, even if $t_i/(m_i\langle \mathsf{left}\rangle)$ is undefined, whereas $q_i\hat{}.left$ is only well defined in a state corresponding to $Is\_Node(t_i/m_i)$.

The right extension operation is similarly transformed:

$$\mathcal{S}[\![\ \{Is\_Node(t_i/m_i)\}\ m_i := m_i\langle \mathsf{right}\rangle\ ]\!] \;\hat{=}\; q_i := q_i\hat{}.right.$$

[2] At the moment, $Is\_Node(t)$ is the same as $t \neq \epsilon$. In Section 5, the definition of trees is extended so this definition of Is_Node is required.

Updating Node Items

Updating of the root of a tree is transformed to an assignment to the root component of the corresponding record:

$$\mathcal{S}[\![\ t_i := t_i \triangleright E\ ]\!] \quad \hat{=} \quad p_i\hat{\ }.root := \mathcal{E}[\![\ E\ ]\!].$$

Similarly for updating the root of a subtree:

$$\mathcal{S}[\![\ t_i := t_i[m_i \triangleright E]\ ]\!] \quad \hat{=} \quad q_i\hat{\ }.root := \mathcal{E}[\![\ E\ ]\!].$$

In both cases, E may depend on t_i, m_i, e.g., $t_i := t_i \triangleright (root(t_i) * 2)$.

3.2 Local variables and procedure calling

A block with local variables is transformed by replacing the tree and path variables with pointers, leaving other variables unchanged and transforming the body:

$$\mathcal{S}[\![\ |[\mathbf{var}\ t_{I'}, m_{J'}, y \bullet S]|\ ]\!] \quad \hat{=} \quad |[\mathbf{var}\ p_{I'}, q_{J'}, y \bullet \mathcal{S}[\![\ S\ ]\!]\ ]|.$$

Assume that a procedure call is of the form

$$\{pre\}\ Proc(\ V_1, \ldots, V_n,\ t_{I'},\ m_{J'},\ E_1, \ldots, E_{n'},\ y\ ),$$

where $V_1, \ldots, V_n$ are tree and path arguments for **val** parameters, $t_{I'}$ and $m_{J'}$ are tree and path arguments for **ref** parameters, $E_1, \ldots, E_{n'}$ are arguments for **val** parameters of a type other than trees or paths and y represents arguments for **ref** parameters of a type other than trees or paths. To avoid aliasing, we assume that no actual variable appears more than once in the lists $t_{I'}$, $m_{J'}$ and y. We also assume that pre is a list of conjuncts each of the form $Is\_Node(t_i)$. These are required to ensure the validity of certain parameter expressions (see below). Such a procedure call is transformed as follows:

$$\begin{array}{l}\mathcal{S}[\![\ \{pre\}\ Proc(\ V_1, \ldots, V_n,\ t_{I'},\ m_{J'},\ E_1, \ldots, E_{n'},\ y\ )\ ]\!] \quad \hat{=} \\ \qquad\qquad Proc(\ \mathcal{V}[\![\ V_1\ ]\!], \ldots, \mathcal{V}[\![\ V_n\ ]\!],\ p_{I'},\ q_{J'},\ \mathcal{E}[\![\ E_1\ ]\!], \ldots, \mathcal{E}[\![\ E_{n'}\ ]\!],\ y\ ).\end{array}$$

The transformation $\mathcal{V}$ on tree arguments is defined as follows:

$$\begin{array}{lcl}\mathcal{V}[\![\ t_i\ ]\!] & \hat{=} & p_i \\ \mathcal{V}[\![\ left(t_i)\ ]\!] & \hat{=} & p_i\hat{\ }.left \\ \mathcal{V}[\![\ right(t_i)\ ]\!] & \hat{=} & p_i\hat{\ }.right.\end{array}$$

Furthermore, if t_i is in the set of **ref** parameters $t_{I'}$, then the procedure may have m_i as a **val** parameter and arguments for m_i are transformed as follows:

$$\begin{array}{lcll}\mathcal{V}[\![\ \langle\rangle\ ]\!] & \hat{=} & p_i & \\ \mathcal{V}[\![\ \langle \mathsf{left} \rangle\ ]\!] & \hat{=} & p_i\hat{\ }.left & \text{provided } Is\_Node(t_i) \in pre \\ \mathcal{V}[\![\ \langle \mathsf{right} \rangle\ ]\!] & \hat{=} & p_i\hat{\ }.right & \text{provided } Is\_Node(t_i) \in pre \\ \mathcal{V}[\![\ m_i\ ]\!] & \hat{=} & q_i & \\ \mathcal{V}[\![\ m_i\langle \mathsf{left} \rangle\ ]\!] & \hat{=} & q_i\hat{\ }.left & \text{provided } Is\_Node(t_i/m_i) \in pre \\ \mathcal{V}[\![\ m_i\langle \mathsf{right} \rangle\ ]\!] & \hat{=} & q_i\hat{\ }.right & \text{provided } Is\_Node(t_i/m_i) \in pre.\end{array}$$

The set of transformable **val** expressions is reiterated in Figure 6.

val Arguments:

t_i	$\langle\rangle$	m_i
$\mathit{left}(t_i)$	$\langle \mathsf{left} \rangle$	$m_i\langle \mathsf{left} \rangle$
$\mathit{right}(t_i)$	$\langle \mathsf{right} \rangle$	$m_i\langle \mathsf{right} \rangle$

Figure 6: Transformable **val** arguments.

Restrictions

As well as the above restriction on procedure calls that avoids aliasing, we need to place some further restrictions on parameter passing when paths are involved. Consider, for example, the following procedure:

$$\textbf{procedure } \mathit{Empty}(\textbf{ref } t_1) \;\hat{=}\; t_1 := \epsilon.$$

If this procedure is called on a tree t_2 which has a path m_2 in scope, then the relationship between t_2 and m_2 may be destroyed as a result. To prevent situations such as this, we impose the constraint that if t_i has path m_i in scope, then

- m_i must be passed as a parameter whenever t_i is and, furthermore,
- t_i must be passed using call by reference.

To see why the second restriction is necessary, consider a procedure declaration of the form **procedure** $\mathit{Proc}(\textbf{val } t_1, m_1) \;\hat{=}\; S$. Within this procedure, we expect that $m_1 \in \mathit{paths}(t_1)$, but this need not be the case if it is called, for example, as $\mathit{Proc}(\mathit{left}(t_2), m_2)$, since $m \in \mathit{paths}(t)$ does not imply that $m \in \mathit{paths}(\mathit{left}(t))$.

To summarise, we allow calls to procedures of the following form:

$$\begin{array}{l}
\textbf{procedure } \mathit{Proc}(\ldots \textbf{val } t_i \ldots) \\
\textbf{procedure } \mathit{Proc}(\ldots \textbf{ref } t_i \ldots) \\
\textbf{procedure } \mathit{Proc}(\ldots \textbf{ref } t_i, \textbf{val } m_i \ldots) \\
\textbf{procedure } \mathit{Proc}(\ldots \textbf{ref } t_i, \textbf{ref } m_i \ldots).
\end{array}$$

Calls to the first two of these are only allowed when the tree used to substitute for t_i does not have a path in scope.

Note that these constraints were not arrived at in an ad-hoc manner but rather arose when considering the soundness of the transformations which is dealt with in Section 8.

4 From Applicative to Imperative I

This section describes some techniques for transforming tree operations specified in an applicative style into imperative-style operations involving paths. These imperative operations are then in a form suitable for transformation to pointer implementations using the pointer-introduction transformation described in Section 3.

4.1 Recursion

Consider the following applicative-style specification of the function that sums all the values in a tree of numbers:

$$\begin{array}{lcl}
\mathit{sum}(\epsilon) & = & 0 \\
\mathit{sum}(\mathsf{node}(n, L, R)) & = & n + \mathit{sum}(L) + \mathit{sum}(R).
\end{array}$$

An imperative procedure that sums the elements of a tree is specified as:

$$\textbf{procedure } Sum(\textbf{val } t, \textbf{ref } y) \;\hat{=}\; y := sum(t).$$

We would expect to be able to refine this with a recursive procedure that calls itself on the left and right subtrees. This can be achieved using the following inference rule for refining a recursive function on trees (REC) with a recursive procedure ($S \sqsubseteq T$ stands for S is refined by T):

Refinement of Recursive Application REC:

$$\begin{array}{lcl} REC(\epsilon) & = & E \\ REC(\mathsf{node}(b, L, R)) & = & F(b, REC(L), REC(R)\,) \end{array}$$

$$\begin{array}{ll}
 & \textbf{procedure } Proc(\textbf{val } t, \textbf{ref } y) \;\hat{=} \\
 & \quad y := REC(t) \\
\sqsubseteq & \\
 & \textbf{procedure } Proc(\textbf{val } t, \textbf{ref } y) \;\hat{=} \\
 & \quad \textbf{if } t = \epsilon \rightarrow \; y := E \\
 & \quad [] \;\; t \neq \epsilon \rightarrow \; |[\textbf{var } y_1, y_2 \bullet \\
 & \qquad\qquad\qquad Proc(left(t), y_1); \;\; Proc(right(t), y_2); \\
 & \qquad\qquad\qquad y := F(root(t), y_1, y_2) \;]| \\
 & \quad \textbf{fi}.
\end{array}$$

The definition of sum is a special case of REC with

$$\begin{array}{rcl} E & = & 0 \\ F(b, E_1, E_2) & = & b + E_1 + E_2. \end{array}$$

According to the above rule, the Sum procedure is refined by:

$$\begin{array}{l}
\textbf{procedure } Sum(\textbf{val } t, \; \textbf{ref } y) \;\hat{=} \\
\quad \textbf{if } t = \epsilon \rightarrow y := 0 \\
\quad [] \;\; t \neq \epsilon \rightarrow |[\textbf{var } y_1, y_2 \bullet \\
\qquad\qquad\qquad Sum(left(t), y_1); \;\; Sum(right(t), y_2); \\
\qquad\qquad\qquad y := root(t) + y_1 + y_2 \;]| \\
\quad \textbf{fi}.
\end{array}$$

Applying the pointer-introduction transformation then results in:

$$\begin{array}{l}
\textbf{procedure } Sum(\textbf{val } p, \; \textbf{ref } y) \;\hat{=} \\
\quad \textbf{if } p = \mathsf{nil} \rightarrow y := 0 \\
\quad [] \;\; p \neq \mathsf{nil} \rightarrow |[\textbf{var } y_1, y_2 \bullet \\
\qquad\qquad\qquad Sum(p\hat{}.left, y_1); \;\; Sum(p\hat{}.right, y_2); \\
\qquad\qquad\qquad y := (p\hat{}.root) + y_1 + y_2 \;]| \\
\quad \textbf{fi}.
\end{array}$$

The recursive structure of this pointer implementation resembles the definition of the sum function and it was arrived at in a straightforward manner using the refinement rule just introduced and the pointer-introduction transformation.

4.2 Mapping

The function REC represents a very general class of recursive definitions. A more specific function on trees is the map function (MAP) which has the following generic form:

Generic Map (MAP):

$$\begin{array}{lcl} MAP(\epsilon) & = & \epsilon \\ MAP(\mathsf{node}(b, L, R)) & = & \mathsf{node}(G(b), MAP(L), MAP(R)). \end{array}$$

The generic map has the following imperative implementation:

Imperative version of Generic Map:

$$t := MAP(t) \quad \sqsubseteq \quad Map(t, \langle\rangle)$$
where
$$\begin{array}{l} \textbf{procedure } Map(\textbf{ref } t,\ \textbf{val } m) \hat{=} \\ \quad \textbf{if } t/m = \epsilon \to \textbf{skip} \\ \quad [\!]\ t/m \neq \epsilon \to \\ \quad\quad t := t[m \triangleright G(root(t/m))]; \\ \quad\quad \{Is\_Node(t/m)\}\ Map(t, m\langle \mathsf{left}\rangle); \\ \quad\quad \{Is\_Node(t/m)\}\ Map(t, m\langle \mathsf{right}\rangle) \\ \quad \textbf{fi}. \end{array}$$

The use of the assertion $\{Is\_Node(t/m)\}$ in the above procedure body is not intended to specify that the program must abort if $Is\_Node(t/m)$ does not hold at that point in the computation. Rather it is a way of encoding an assumption that can be made use of in transforming the subsequent calls to *Map* (recall that the transformation rule for **val** parameters in procedure calls requires some preconditions). The assumption is valid at that point because of the guard $t/m \neq \epsilon$.

Applying the pointer-introduction transformation to the above procedure results in:

$$\begin{array}{l} \textbf{procedure } Map(\textbf{ref } p,\ \textbf{val } q) \hat{=} \\ \quad \textbf{if } q = \mathsf{nil} \to \textbf{skip} \\ \quad [\!]\ q \neq \mathsf{nil} \to \\ \quad\quad q\hat{\ }.root := G(q\hat{\ }.root); \\ \quad\quad Map(p, q\hat{\ }.left); \\ \quad\quad Map(p, q\hat{\ }.right) \\ \quad \textbf{fi}. \end{array}$$

At the pointer level, the parameter p is redundant and may be removed. The use of a path in the abstract imperative procedure provides a close match with the pointer implementation. Instead of calling *Map* on $left(t)$ and $right(t)$ and then reconstructing the tree with the results, we simply pass around a path indicating which part of t is to be operated on in a recursive call; this path is implemented by a pointer.

Note that the *Sum* and *Map* procedures satisfy the constraints on parameters described in Section 3.2.

4.3 Directed Search

Another specific function on trees is the *directed search* (*SRCH*) which has the following generic form:

Generic Directed Search ($SRCH$):

$$
\begin{array}{lcl}
SRCH(\epsilon) & = & E1 \\
SRCH(\mathsf{node}(b, L, R)) & = & \textbf{if } CT(\mathsf{node}(b, L, R)) \rightarrow E2(\mathsf{node}(b, L, R)) \\
 & & [\!]\ CL(\mathsf{node}(b, L, R)) \rightarrow SRCH(L) \\
 & & [\!]\ CR(\mathsf{node}(b, L, R)) \rightarrow SRCH(R) \\
 & & \textbf{fi}.
\end{array}
$$

Where for any t s.t. $Is\_Node(t)$, $CT(t)$, $CL(t)$ and $CR(t)$ are mutually exclusive and exhaustive.

The search is directed by the guards CL and CR and ends when an empty subtree is reached or when CT holds. The (curried) *member* function (see Figure 1) is a special case of $SRCH$, with

$$
\begin{array}{rcl}
E1 & = & false \\
CT(t) & = & a = root(t) \\
E2(t) & = & true \\
CL(t) & = & a < root(t) \\
CR(t) & = & a > root(t).
\end{array}
$$

A recursive implementation of $x := SRCH(t)$ can easily be derived using the recursion-refinement rule of Section 4.1. Because of the guarding in the second clause of $SRCH$, there is only ever one recursive call. This means that $x := SRCH(t)$ can also be implemented by a loop. In this case, the refinement template is as follows:

Imperative version of Generic Directed Search:

$$
\begin{array}{ll}
 & x := SRCH(t) \\
\sqsubseteq & \\
 & |[\textbf{var } m \bullet m := \langle\rangle; \\
 & \textbf{do } t/m \neq \epsilon \ \wedge\ \neg CT(t/m) \rightarrow \\
 & \quad \textbf{if } CL(t/m) \rightarrow \{Is\_Node(t/m)\}\ m := m\langle\mathsf{left}\rangle \\
 & \quad [\!]\ CR(t/m) \rightarrow \{Is\_Node(t/m)\}\ m := m\langle\mathsf{right}\rangle\ \textbf{fi} \\
 & \textbf{od}; \\
 & \textbf{if } t/m = \epsilon \rightarrow x := E1 \\
 & [\!]\ t/m \neq \epsilon \rightarrow x := E2(t/m) \\
 & \textbf{fi}\ ]|.
\end{array}
$$

The use of paths again simplifies the treatment by avoiding aliasing. We could have implemented $x := SRCH(t)$ by something like:

$$
\begin{array}{l}
|[\textbf{var } t_2 \bullet \\
\quad t_2 := t; \\
\quad \textbf{do } t_2 \neq \epsilon \ \wedge\ \neg CT(t_2) \rightarrow \\
\quad\quad \textbf{if } CL(t_2) \rightarrow t_2 := left(t_2) \cdots
\end{array}
$$

However, the initial assignment $t_2 := t$ would either have to be implemented by copying t (an unsatisfactory solution) or we would have t and t_2 aliased to the same structure. Aliasing complicates the proof system so we avoid it. This issue is discussed further in Section 5.

Using the above rule, $x := member(a)(t)$ is refined by:

$$
\begin{array}{l}
|[\textbf{var } m \bullet m := \langle\rangle; \\
\textbf{do } t/m \neq \epsilon \ \wedge \ a \neq root(t/m) \rightarrow \\
\quad \textbf{if } a < root(t/m) \rightarrow \{Is\_Node(t/m)\} \ m := m\langle \mathsf{left} \rangle \\
\quad [] \ \ a > root(t/m) \rightarrow \{Is\_Node(t/m)\} \ m := m\langle \mathsf{right} \rangle \ \textbf{fi} \\
\textbf{od}; \\
\textbf{if } t/m = \epsilon \rightarrow x := false \\
[] \ \ t/m \neq \epsilon \rightarrow x := true \\
\textbf{fi} \ ]|.
\end{array}
$$

The resulting pointer implementation is shown in Figure 3 (where the final if-statement has been simplified to a single assignment).

Another instance of the generic search scheme is the *lookup* function that returns the data associated with a particular key, where items are ordered by keys. Assume

$$Item \ \hat{=} \ \textbf{record } key : Key, \ data : Data \ \textbf{end},$$

and that $lookup(k)$ returns values from the disjoint union type $ok(Data)|error$. Now $lookup(k)$ is a special case of $SRCH$ with

$$
\begin{array}{rcl}
E1 & = & error \\
CT(t) & = & k = root(t).key \\
E2(t) & = & ok(root(t).data) \\
CL(t) & = & k < root(t).key \\
CR(t) & = & k > root(t).key.
\end{array}
$$

4.4 Correctness

The correctness of the refinement rules for REC and MAP is easily verified using the following refinement rule for recursion introduction:

$$\frac{\{v = v_0\} \ S \ \sqsubseteq \ T[\ \{v < v_0\} \ S \ ]}{\textbf{procedure } Proc \ \hat{=} \ S \ \sqsubseteq \ \textbf{procedure } Proc \ \hat{=} \ T[Proc].}$$

Here, $T[Proc]$ is a statement with one or more occurrences of a recursive call to $Proc$ and $T[S]$ is that statement with all occurrences of the recursive call replaced by S (see Section 8.2 for a description of how parameters are dealt with). Also, v is a variant expression which must be decreased in each recursive call. For REC, the size of the tree acts as a variant since $size(left(t)) < size(t)$ etc. For MAP the variant is $size(t/m)$.

The correctness of the refinement rule for $SRCH$ is easily verified using the standard loop-verification rule which requires an invariant and a variant. The invariant in this case is

$$m \in paths(t) \ \wedge \ SRCH(t) = SRCH(t/m),$$

while the variant is $size(t/m)$.

5 The Arbitrary Tree

In this section, we introduce 'cut' and 'paste' operations on abstract trees that correspond to commonly-used manipulations of pointer structures.

We regard $t_i := t_i[m_i \backslash t_j]$ as a form of paste operation. One way to implement this would be to make a copy of t_j and update the parent node of q_i (q_i is the pointer to t_i/m_i) so that it points to this new copy of t_j. We want to avoid having to make a copy of t_j and, instead, have an abstract representation of the pointer operation that makes the parent node of q_i point directly to the representation of t_j. However, allowing t_j to be an alias for a substructure of t_i would have the consequence that any later assignment to t_j would have the side effect of changing the concrete representation of t_i also. Allowing such aliasing would complicate the proof system so, to avoid it, we introduce a special tree, the $*$-tree or arbitrary tree, that is distinct from ϵ and $\mathsf{node}(a, L, R)$:

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

The intention is 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 $\mathit{Match}(t, p, \sigma)$ which is true if and only if the pointer structure in store σ starting at pointer p is a valid representation of tree t (recall our assumption that $\mathsf{nil} \notin \mathit{dom}(\sigma)$):

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

The difference between ϵ and $*$ can be clearly seen: $*$ will match with any structure, while ϵ will only match with nil. Note that we do not require that all pointers be well defined in σ, e.g., $*$ will match with p even if $p \neq \mathsf{nil} \wedge p \notin \mathit{dom}(\sigma)$. Such situations may arise when the **dispose** command is used resulting in so-called "dangling" pointers. In the next section, we present a transformation for introducing **dispose** in a way that never results in harmful dangling pointers.

Now, the paste operation that makes the parent node of q_i point directly to the representation of t_j is represented abstractly as:

$$t_i, t_j := t_i[m_i \backslash t_j], *.$$

This can be thought of as simultaneously 'cutting' the entire contents of t_j and 'pasting' it at the position of t_i pointed to by m_i.

We also introduce the *cut* operator which replaces a sub-tree of t with the $*$-tree:

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

An operation which cuts a subtree of t_i and places it in t_j may be defined as follows:

$$t_i, t_j := t_i \upharpoonright m_i, t_i/m_i.$$

This operation is implemented at the pointer level simply as:

$$p_j := q_i.$$

This is a valid implementation because, although p_j now points to a sub-structure of p_i, we no longer care about this subtree of t_i at the abstract level since it has become the $*$-tree. Any later assignments to the representation of t_j will not have any side effect on the representation of t_i. This is formalised in Section 8.

Note that our definitions of t/m and $t_i[m_i \backslash t_j]$ remain valid with our extended definition of trees. We do need to extend the definition of $\mathit{paths}(t)$. Since $*/\langle\rangle$ and $*[\langle\rangle \backslash t]$ are valid expressions, we have:

$$\mathit{paths}(*) \quad \hat{=} \quad \{\ \langle\rangle\ \}.$$

We also note the following equalities:

$$\begin{array}{rcl} m_i \in paths(t_i) \Rightarrow & & \\ (t_i[m_i\backslash t_j])/m_i & = & t_j \\ t_i[m_i\backslash(t_i/m_i)] & = & t_i \\ (t_i[m_i\backslash t_j])[m_i\backslash t_k] & = & t_i[m_i\backslash t_k] \\ (t_i \restriction m_i)[m_i\backslash t_j] & = & t_i[m_i\backslash t_j]. \end{array}$$

6 Pointer-Introduction Transformation II

As already mentioned in the previous section, in order to implement an operation of the form $t_i, t_j := t_i[m_i\backslash t_j], *$, we need to update the parent node of q_i so that it points to the representation of t_j. For this, an extra pointer r_i, modelling the parent of q_i is introduced. We make a distinction between paths that are used in paste operations $t_i[m_i\backslash t_j]$, which we refer to as *pasting paths*, and those that are only used to access subtrees t_i/m_i. Making this distinction avoids the introduction of too many redundant pointers as non-pasting paths need only be replaced by one pointer rather than two. We write $\overline{m}_i$ to indicate that a transformation only applies to a pasting path.

All of the transformations of Section 3 apply to specifications involving pasting paths except for the following important differences:

1. Each abstract pasting path $\overline{m}_i$ in a module declaration, in a local variable declaration or in a call-by-reference (**ref**) parameter list is transformed to two pointers, q_i and r_i.

2. A pasting path cannot be used as a call-by-value (**val**) parameter. The reason for this restriction will be made clear later.

3. The operations for making a left or a right move of a pasting path are transformed as follows:

$$\begin{array}{lcl} \mathcal{S}[\![\ \{Is\_Node(t_i/\overline{m}_i)\}\ \overline{m}_i := \overline{m}_i\langle \mathsf{left}\rangle\ ]\!] & \hat{=} & q_i,\ r_i := q_i\hat{}.left,\ q_i \\ \mathcal{S}[\![\ \{Is\_Node(t_i/\overline{m}_i)\}\ \overline{m}_i := \overline{m}_i\langle \mathsf{right}\rangle\ ]\!] & \hat{=} & q_i,\ r_i := q_i\hat{}.right,\ q_i. \end{array}$$

Furthermore, all of the applicative-to-imperative transformation schemes presented in Section 4 also remain valid provided the tree variables in the abstract specifications do not contain any occurrences of the $*$-tree. This is formalised by preceding each abstract specification with the assertion $\{Is\_Whole(t)\}$, where

$$Is\_Whole(t) \quad \hat{=} \quad \forall m \in paths(t) \cdot t/m \neq *.$$

For example, the consequent of the rule for refinement of recursive application (page 11) becomes:

$$\begin{array}{ll} & \textbf{procedure}\ Proc(\textbf{val}\ t, \textbf{ref}\ y)\ \hat{=} \\ & \quad \{Is\_Whole(t)\}\ y := REC(t) \\ \sqsubseteq & \\ & \textbf{procedure}\ Proc(\textbf{val}\ t, \textbf{ref}\ y)\ \hat{=} \\ & \quad \textbf{if}\ t = \epsilon \rightarrow\ y := E \\ & \quad [\!]\ t \neq \epsilon \rightarrow\ |[\textbf{var}\ y_1, y_2 \bullet \\ & \qquad\qquad Proc(left(t), y_1);\ \ Proc(right(t), y_2); \\ & \qquad\qquad y := F(root(t), y_1, y_2)\ ]| \\ & \quad \textbf{fi}. \end{array}$$

$$
\boxed{\begin{array}{l} root(t_i/front(\overline{m}_i)) \\ root(left(t_i/front(\overline{m}_i))) \\ root(right(t_i/front(\overline{m}_i))) \end{array}}
$$

Figure 7: More transformable tree expressions.

There are also some additional transformations for expressions involving pasting paths :

$$
\begin{array}{lcl}
\mathcal{E}[\![\ root(t_i/front(\overline{m}_i))\ ]\!] & \hat{=} & r_i\hat{\ }.root \\
\mathcal{E}[\![\ root(left(t_i/front(\overline{m}_i)))\ ]\!] & \hat{=} & (r_i\hat{\ }.left)\hat{\ }.root \\
\mathcal{E}[\![\ root(right(t_i/front(\overline{m}_i)))\ ]\!] & \hat{=} & (r_i\hat{\ }.right)\hat{\ }.root.
\end{array}
$$

Here, $front(m)$ is all of path m except its last element. These additional transformable tree expressions are reiterated in Figure 7.

6.1 Cutting

The operation that cuts a tree in two using a path is transformed to a simple pointer assignment as follows: provided t_j has no associated path, then

$$
\mathcal{S}[\![\ t_i,\ t_j := t_i \restriction m_i,\ t_i/m_i\ ]\!] \quad \hat{=} \quad p_j := q_i.
$$

Operations for cutting away the left and right subtrees are also transformable:

$$
\begin{array}{lcl}
\mathcal{S}[\![\ t_i,\ t_j := t_i \restriction \mathsf{left},\ t_i/\mathsf{left}\ ]\!] & \hat{=} & p_j := p_i\hat{\ }.left \\
\mathcal{S}[\![\ t_i,\ t_j := t_i \restriction \mathsf{right},\ t_i/\mathsf{right}\ ]\!] & \hat{=} & p_j := p_i\hat{\ }.right \\
\mathcal{S}[\![\ t_i,\ t_j := t_i \restriction (m_i\langle \mathsf{left}\rangle),\ t_i/(m_i\langle \mathsf{left}\rangle)\ ]\!] & \hat{=} & p_j := q_i\hat{\ }.left \\
\mathcal{S}[\![\ t_i,\ t_j := t_i \restriction (m_i\langle \mathsf{right}\rangle),\ t_i/(m_i\langle \mathsf{right}\rangle)\ ]\!] & \hat{=} & p_j := q_i\hat{\ }.right.
\end{array}
$$

Here, $t_i \restriction \mathsf{left}$ is short for $t_i \restriction \langle \mathsf{left}\rangle$, etc.

6.2 Pasting

An implementation of a paste operation of the form

$$
t_i,\ t_j := t_i[\overline{m}_i \backslash t_j],\ *
$$

needs to update the node pointed to by r_i, i.e., the parent node of q_i. Whether it is the right or left field of this node that needs to be updated will depend on the last value in the path $\overline{m}_i$ ($last(\overline{m}_i)$). However, since $\overline{m}_i$ is removed in the transformation, we need to supply two predicates L and R such that

$$
\begin{array}{lcl}
L & \Rightarrow & last(\overline{m}_i) = \mathsf{left} \\
R & \Rightarrow & last(\overline{m}_i) = \mathsf{right}
\end{array}
$$

and L and R are transformable by $\mathcal{P}$. In Section 7, we will see how L and R may be derived using a refinement template.

Transformation of the paste operation is defined as follows: provided t_j has no associated path, then

$$
\begin{array}{l}
\mathcal{S}[\![\ \{\ P\ \}\ \ t_i,\ t_j := t_i[\overline{m}_i \backslash t_j],\ *\ ]\!]\ \hat{=} \\
\qquad \textbf{if}\ \mathcal{E}[\![\ L\ ]\!] \rightarrow r_i\hat{\ }.left, q_i := p_j, p_j \\
\qquad [\!]\ \ \mathcal{E}[\![\ R\ ]\!] \rightarrow r_i\hat{\ }.right, q_i := p_j, p_j \\
\qquad \textbf{fi},
\end{array}
$$

where, for some L and R, P has the form:

$$\begin{array}{rcl} P & \equiv & \overline{m}_i \neq \langle\rangle \wedge\ (L \vee R) \wedge \\ & & L \Rightarrow last(\overline{m}_i) = \mathsf{left}\ \ \wedge \\ & & R \Rightarrow last(\overline{m}_i) = \mathsf{right}. \end{array}$$

Note that, although the abstract paste operation does not change the path $\overline{m_i}$, its concrete implementation does make an assignment to the pointer q_i. Since **val** parameters cannot be assigned to, this is the reason for our restriction that a pasting path cannot be used as a **val** parameter in a procedure.

We also have the following transformations:

$$\begin{array}{lcl} \mathcal{S}[\![\ t_i, t_j := t_j, *\ ]\!] & \hat{=} & p_i := p_j \\ \mathcal{S}[\![\ t_i, t_j := t_i[\mathsf{left}\backslash t_j], *\ ]\!] & \hat{=} & p_i\hat{\ }.left := p_j \\ \mathcal{S}[\![\ t_i, t_j := t_i[\mathsf{right}\backslash t_j], *\ ]\!] & \hat{=} & p_i\hat{\ }.right := p_j \\ \mathcal{S}[\![\ t_i, t_j := t_i[m_i\langle\mathsf{left}\rangle\backslash t_j], *\ ]\!] & \hat{=} & q_i\hat{\ }.left := p_j \\ \mathcal{S}[\![\ t_i, t_j := t_i[m_i\langle\mathsf{right}\rangle\backslash t_j], *\ ]\!] & \hat{=} & q_i\hat{\ }.right := p_j. \end{array}$$

6.3 Constructors and Destructors

Construction of a new node with subtrees t_j and t_k is transformed as follows: provided t_j and t_k have no associated paths and E is independent of t_i, then

$$\begin{array}{l} \mathcal{S}[\![\ t_i,\ t_j,\ t_k := \mathsf{node}(E, t_j, t_k),\ *,\ \ *\ ]\!]\ \hat{=} \\ \qquad\qquad \mathbf{new}(p_i);\ \ p_i\hat{\ }.root,\ p_i\hat{\ }.left,\ p_i\hat{\ }.right := \mathcal{E}[\![\ E\ ]\!],\ p_j,\ p_k. \end{array}$$

So far we have not provided a transformation that destructs a node using the **dispose** statement. It is only safe to deallocate a record pointed to by p_i if we know that record is no longer required. At the abstract level, no longer requiring the root node of a tree is modelled by the statement:

$$\{Is\_Node(t_i)\}\ t_i := \ *\ .$$

The assertion says that we know t_i is a node, so the root node will be represented at the pointer level by a valid record, and the assignment says that we no longer care about the value of t_i. Since sharing of nodes by trees cannot occur (modulo $*$), t_i is the only tree whose root node is represented by the record pointed to by p_i so that record may be safely removed. Thus we have

$$\mathcal{S}[\![\ \{Is\_Node(t_i)\}\ t_i := \ *\ ]\!]\ \ \hat{=}\ \ \mathbf{dispose}(p_i).$$

Of course, this will only deallocate the record representing the root of the tree, not its descendants, so it would only be appropriate to apply it when $left(t_i) = \epsilon\ \vee\ left(t_i) = *$, and similarly for $right(t_i)$.

The additional transformable tree assignments introduced in this section are reiterated in Figure 8. It is best that a $*$-tree only be assigned to a tree variable when that tree will later be pasted to or when that tree is a local variable about to go out of scope. Similarly, it is best that a $*$-tree only be assigned to part of a tree when that part will later be pasted to or when that part is a node that may be disposed of. These guidelines help ensure that a tree that starts off being whole does not unintentionally end up containing the $*$-tree.

7 From Applicative to Imperative II

A variation on the directed search presented in Section 4 is the *directed update* which replaces a subtree reached through a directed search, leaving the rest of the

$$
\begin{array}{ll}
t_i,\ t_j := t_i \restriction m_i,\ t_i / m_i & t_i,\ t_j := t_i \restriction \mathsf{left},\ t_i / \mathsf{left} \\
t_i,\ t_j,\ t_k := \mathsf{node}(b, t_j, t_k),\ *,\ * & t_i,\ t_j := t_i \restriction \mathsf{right},\ t_i / \mathsf{right} \\
t_i,\ t_j := t_j,\ * & t_i,\ t_j := t_i \restriction m_i\langle \mathsf{left} \rangle,\ t_i / m_i\langle \mathsf{left} \rangle \\
\{Is\_Node(t_i)\}\ t_i := * & t_i,\ t_j := t_i \restriction m_i\langle \mathsf{right} \rangle,\ t_i / m_i\langle \mathsf{right} \rangle \\
\{\ P\ \}\ \ t_i,\ t_j := t_i[\overline{m}_i \backslash t_j],\ * & t_i,\ t_j := t_i[\mathsf{left} \backslash t_j],\ * \\
\text{where} & t_i,\ t_j := t_i[\mathsf{right} \backslash t_j],\ * \\
\quad P \ \equiv \ \overline{m}_i \neq \langle\rangle \wedge\ (L \vee R) \wedge & t_i,\ t_j := t_i[m_i\langle \mathsf{left} \rangle \backslash t_j],\ * \\
\qquad\qquad L \Rightarrow last(\overline{m}_i) = \mathsf{left}\ \wedge & \\
\qquad\qquad R \Rightarrow last(\overline{m}_i) = \mathsf{right} & t_i,\ t_j := t_i[m_i\langle \mathsf{right} \rangle \backslash t_j],\ *
\end{array}
$$

Figure 8: More transformable tree assignments.

tree intact:

Generic Directed Update (UPD):

$$
\begin{array}{lcl}
UPD(\epsilon) & = & E1 \\
UPD(\mathsf{node}(b, L, R)) & = & \textbf{if}\ CT(\mathsf{node}(b, L, R)) \rightarrow E2(\mathsf{node}(b, L, R)) \\
& & []\ \ CL(\mathsf{node}(b, L, R)) \rightarrow \mathsf{node}(b, UPD(L), R) \\
& & []\ \ CR(\mathsf{node}(b, L, R)) \rightarrow \mathsf{node}(b, L, UPD(R)) \\
& & \textbf{fi}.
\end{array}
$$

Where for any t s.t. $Is\_Node(t)$, $CT(t)$, $CL(t)$ and $CR(t)$ are mutually exclusive and exhaustive.

The insert function, which inserts an item in the appropriate place in an ordered tree, is an instance of UPD:

$$
\begin{array}{lcl}
insert(a)(\epsilon) & = & \mathsf{node}(a, \epsilon, \epsilon) \\
insert(a)(\mathsf{node}(b, L, R)) & = & \textbf{if}\ a = b \rightarrow \mathsf{node}(b, L, R) \\
& & []\ \ a < b \rightarrow \mathsf{node}(b, insert(a)(L), R) \\
& & []\ \ a > b \rightarrow \mathsf{node}(b, L, insert(a)(R)) \\
& & \textbf{fi}.
\end{array}
$$

In this case, we have

$$
\begin{array}{rcl}
E1 & = & \mathsf{node}(a, \epsilon, \epsilon) \\
CT(t) & = & a = root(t) \\
E2(t) & = & t \\
CL(t) & = & a < root(t) \\
CR(t) & = & a > root(t).
\end{array}
$$

The directed update has the following imperative implementation:

Imperative version of Generic Directed Update:

$$
\begin{array}{l}
\quad \{\mathit{Is\_Whole}(t)\}\ t := \mathit{UPD}(t) \\
\sqsubseteq \\
\quad \textbf{if } t = \epsilon \rightarrow t := E1 \\
\quad [\!]\ \ t \neq \epsilon\ \wedge\ CT(t) \rightarrow t := E2(t) \\
\quad [\!]\ \ t \neq \epsilon\ \wedge\ \neg CT(t) \rightarrow \\
\qquad |[\textbf{var } m \bullet m := \langle\rangle; \\
\qquad \textbf{do } t/m \neq \epsilon\ \wedge\ \neg CT(t/m) \rightarrow \\
\qquad\quad \textbf{if } CL(t/m) \rightarrow \{\mathit{Is\_Node}(t/m)\}\ m := m\langle\mathsf{left}\rangle \\
\qquad\quad [\!]\ \ CR(t/m) \rightarrow \{\mathit{Is\_Node}(t/m)\}\ m := m\langle\mathsf{right}\rangle\ \textbf{fi} \\
\qquad \textbf{od}; \\
\qquad \{\ m \in \mathit{paths}(t)\ \wedge\ m \neq \langle\rangle\ \wedge \\
\qquad\ \ (CL(t/\mathit{front}(m)) \vee CR(t/\mathit{front}(m)) \wedge \\
\qquad\ \ (CL(t/\mathit{front}(m)) \Rightarrow \mathit{last}(m) = \mathsf{left}) \wedge \\
\qquad\ \ (CR(t/\mathit{front}(m)) \Rightarrow \mathit{last}(m) = \mathsf{right})\ \ \}; \\
\qquad \textbf{if } t/m = \epsilon \rightarrow t := t[m \backslash E1] \\
\qquad [\!]\ \ t/m \neq \epsilon \rightarrow t := t[m \backslash E2(t/m)] \\
\qquad \textbf{fi}\ ]| \\
\quad \textbf{fi}.
\end{array}
$$

Notice the assertion that holds immediately after the loop. This assertion encodes an important assumption about CL and CR that allows them to be used as guards for the refinement of the later paste operations. If we disregard this assertion, then the body of the third branch of the outermost if-statement is equivalent to the full outermost if-statement. This is because the loop body would never be executed when the first two guards held. However, the assertion after the loop (necessarily) contains the clause $m \neq \langle\rangle$ which will only hold if the loop body is executed at least once. The outermost if-statement ensures that the loop body is executed at least once.

We get the following refinement of $\{\mathit{Is\_Whole}(t_1)\}\ t_1 := \mathit{insert}(a)(t_1)$ using the above rule:

$$
\begin{array}{ll}
\textbf{if } t_1 = \epsilon \rightarrow t_1 := \mathsf{node}(a, \epsilon, \epsilon) & \\
[\!]\ \ t_1 \neq \epsilon\ \wedge\ a = \mathit{root}(t_1) \rightarrow t_1 := t_1 & (1) \\
[\!]\ \ t_1 \neq \epsilon\ \wedge\ a \neq \mathit{root}(t_1) \rightarrow & \\
\quad |[\textbf{var } m_1 \bullet m_1 := \langle\rangle; & \\
\quad \textbf{do } t_1/m_1 \neq \epsilon\ \wedge\ a \neq \mathit{root}(t_1/m_1) \rightarrow & \\
\qquad \textbf{if } a < \mathit{root}(t_1/m_1) \rightarrow \{\mathit{Is\_Node}(t_1/m_1)\}\ m_1 := m_1\langle\mathsf{left}\rangle & \\
\qquad [\!]\ \ a > \mathit{root}(t_1/m_1) \rightarrow \{\mathit{Is\_Node}(t_1/m_1)\}\ m_1 := m_1\langle\mathsf{right}\rangle\ \textbf{fi} & \\
\quad \textbf{od}; & \\
\quad \{\ m_1 \in \mathit{paths}(t_1)\ \wedge\ m_1 \neq \langle\rangle\ \wedge & \\
\qquad (a < \mathit{root}(t_1/\mathit{front}(m_1)) \vee a > \mathit{root}(t_1/\mathit{front}(m_1)) \wedge & \\
\qquad (a < \mathit{root}(t_1/\mathit{front}(m_1)) \Rightarrow \mathit{last}(m_1) = \mathsf{left}) \wedge & \\
\qquad (a > \mathit{root}(t_1/\mathit{front}(m_1)) \Rightarrow \mathit{last}(m_1) = \mathsf{right})\ \ \}; & \\
\quad \textbf{if } t_1/m_1 = \epsilon \rightarrow t_1 := t_1[m_1 \backslash \mathsf{node}(a, \epsilon, \epsilon)] & (2) \\
\quad [\!]\ \ t_1/m_1 \neq \epsilon \rightarrow t_1 := t_1[m_1 \backslash (t_1/m_1)] & (3) \\
\quad \textbf{fi}\ ]| & \\
\textbf{fi}. &
\end{array}
$$

The assignments of lines (1), (2) and (3) need to be further refined before the pointer-introduction transformation can be applied. From line (1), $t_1 := t_1 \sqsubseteq$ **skip**.

Let $I2$ be the assertion that follows the loop above. Using rules for propagating assertions[3] [2, 14], the assignment in line (2) can be expanded to:

$$\{I2\}\ t_1 := t_1[m_1 \backslash \mathsf{node}(a, \epsilon, \epsilon)].$$

This is then refined by[4]:

$$|[\ \mathbf{var}\ t_2 \bullet t_2 := \mathsf{node}(a, \epsilon, \epsilon); \quad \{I2\}\ t_1,\ t_2 := t_1[m_1 \backslash t_2],\ *\ ]|,$$

which is in a form that is transformable by $\mathcal{S}$.

The assignment in line (3) can be expanded and refined as follows:

$$\begin{array}{ll} & \{m_1 \in paths(t_1)\}\ t_1 := t_1[m_1 \backslash (t_1/m_1)] \\ \sqsubseteq & \quad \text{“}\ m \in paths(t) \Rightarrow t[m \backslash (t/m)] = t\ \text{”} \\ & t_1 := t_1 \\ \sqsubseteq & \mathbf{skip}. \end{array}$$

The pointer-introduction transformation may now be applied, resulting in:

$$\begin{array}{l}
\mathbf{if}\ p_1 = \mathsf{nil} \rightarrow \mathbf{new}(p_1);\ \ p_1\hat{}.root,\ p_1\hat{}.left,\ p_1\hat{}.right := a,\ \mathsf{nil},\ \mathsf{nil} \\
[]\ \ p_1 \neq \mathsf{nil} \wedge a = p_1\hat{}.root \rightarrow \mathbf{skip} \\
[]\ \ p_1 \neq \mathsf{nil} \wedge a \neq p_1\hat{}.root \rightarrow \\
\quad\quad |[\mathbf{var}\ q_1, r_1 \bullet q_1 := p_1; \\
\quad\quad \mathbf{do}\ q_1 \neq \mathsf{nil} \wedge a \neq q_1\hat{}.root \rightarrow \\
\quad\quad\quad\quad \mathbf{if}\ a < q_1\hat{}.root \rightarrow q_1, r_1 := q_1\hat{}.left, q_1 \\
\quad\quad\quad\quad []\ \ a > q_1\hat{}.root \rightarrow q_1, r_1 := q_1\hat{}.right, q_1 \\
\quad\quad \mathbf{od}; \\
\quad\quad \mathbf{if}\ q_1 = \mathsf{nil} \rightarrow \\
\quad\quad\quad\quad |[\ \mathbf{var}\ p_2 \bullet \\
\quad\quad\quad\quad\quad\quad \mathbf{new}(p_2);\ \ p_2\hat{}.root,\ p_2\hat{}.left,\ p_2\hat{}.right := a,\ \mathsf{nil},\ \mathsf{nil}; \\
\quad\quad\quad\quad\quad\quad \mathbf{if}\ a < r_1\hat{}.root \rightarrow r_1\hat{}.left, q_1 := p_2, p_2 \\
\quad\quad\quad\quad\quad\quad []\ \ a > r_1\hat{}.root \rightarrow r_1\hat{}.right, q_1 := p_2, p_2 \\
\quad\quad\quad\quad\quad\quad \mathbf{fi}\ \ ]| \\
\quad\quad []\ \ q_1 \neq \mathsf{nil} \rightarrow \mathbf{skip} \\
\quad\quad \mathbf{fi}\ ]| \\
\mathbf{fi}.
\end{array}$$

7.1 Tree Deletion

An operation to remove an element from an ordered tree is defined as follows:

$$\begin{array}{lcl}
remove(a)(\epsilon) & = & \epsilon \\
remove(a)(\mathsf{node}(b, L, R)) & = & \mathbf{if}\ a = b \rightarrow join(L)(R) \\
 & & []\ \ a < b \rightarrow \mathsf{node}(b,\ remove(a)(L),\ R) \\
 & & []\ \ a > b \rightarrow \mathsf{node}(b,\ L,\ remove(a)(R))\ \mathbf{fi} \\
 & & \\
join(t)(\epsilon) & = & t \\
join(t)(\mathsf{node}(a, L, R)) & = & \mathbf{if}\ L = \epsilon \rightarrow \mathsf{node}(a,\ t,\ R) \\
 & & []\ \ L \neq \epsilon \rightarrow \mathsf{node}(a,\ join(t)(L),\ R)\ \mathbf{fi}.
\end{array}$$

Since all elements of L are less than all elements of R, $join(L)(R)$ will be ordered. Like the $insert(a)$ function, the $remove(a)$ function is a special case of UPD, with

$$E1 \ = \ \epsilon$$

[3] $\{p \wedge q\} \sqsubseteq \{p\}$
$\{p\}\ \mathbf{if}\ C_1 \rightarrow S_1 [] \cdots []\ C_n \rightarrow S_n\ \mathbf{fi}\ \sqsubseteq\ \mathbf{if}\ C_1 \rightarrow \{p \wedge C_1\} S_1 [] \cdots []\ C_n \rightarrow \{p \wedge C_n\} S_n\ \mathbf{fi}.$

[4] $\{p\}\ x := E(e)\ \ \sqsubseteq\ \ |[\mathbf{var}\ y \bullet y := e;\ \ \{p\}\ x, y := E(y), e']|,\ \ \text{for any } e'.$

$$\begin{array}{rcl} CT(t) & = & a = root(t) \\ E2(t) & = & join(left(t))(right(t)) \\ CL(t) & = & a < root(t) \\ CR(t) & = & a > root(t). \end{array}$$

Using the rule for refining UPD, we get a refinement of the statement $\{Is\_Whole(t_1)\}$ $t_1 := remove(a)(t_1)$ with the following outline form:

$$\begin{array}{l} \textbf{if} \ \ldots \\ [\!]\ \ t_1 \neq \epsilon \ \wedge \ a \neq root(t_1) \rightarrow \\ \qquad |[\textbf{var}\ m_1 \bullet \ldots \\ \qquad \textbf{do}\ \ldots \textbf{od}; \\ \qquad \{\ldots\}; \\ \qquad \textbf{if}\ \ldots \\ \qquad [\!]\ \ t_1/m_1 \neq \epsilon \rightarrow t_1 := t_1[m_1 \backslash join(left(t_1/m_1))(right(t_1/m_1))] \qquad (4) \\ \qquad \textbf{fi}\ ]| \\ \textbf{fi}. \end{array}$$

Continuing with the assignment statement (4), it is easy to show, using standard refinement rules, that

$$\begin{array}{l} \{\ Is\_Whole(t_1/m_1) \ \wedge \ t_1/m_1 \neq \epsilon \ \} \ \ t_1 := t_1[m_1 \backslash join(left(t_1/m_1))(right(t_1/m_1))] \\ \quad \sqsubseteq \\ \qquad |[\textbf{var}\ t_2, t_3, t_4 \bullet \\ \qquad\quad t_1, t_2 := t_1 \upharpoonright m_1,\ t_1/m_1; \\ \qquad\quad t_2, t_3 := t_2 \upharpoonright \mathsf{left},\ t_2/\mathsf{left}; \\ \qquad\quad t_2, t_4 := t_2 \upharpoonright \mathsf{right},\ t_2/\mathsf{right}; \\ \qquad\quad \{Is\_Node(t_2)\}\ t_2 := *; \\ \qquad\quad \{Is\_Whole(t_4)\}\ t_4, t_3 := join(t_3)(t_4),\ *; \qquad (5) \\ \qquad\quad t_1, t_4 := t_1[m_1 \backslash t_4],\ * \\ \qquad]|. \end{array}$$

In the refined block, t_2 becomes t_1/m_1 then t_3 becomes t_2/left and t_4 becomes t_2/right. Next t_3 is joined to t_4 and the result is pasted to t_1.

Each of the statements in the refinement, except (5), is directly transformable by $\mathcal{S}$. For example, the statement $\{Is\_Node(t_2)\}$ $t_2 := *$ is transformed to $\textbf{dispose}(p_2)$. Furthermore, $join(t_3)(t_4)$ is itself a special case of $UPD(t_4)$, with

$$\begin{array}{rcl} E1 & = & t_3 \\ CT(t_4) & = & left(t_4) = \epsilon \\ E2(t_4) & = & \mathsf{node}(root(t_4), t_3, right(t_4)) \\ CL(t_4) & = & left(t_4) \neq \epsilon \\ CR(t_4) & = & false, \end{array}$$

so that (5) can itself be refined using the rule for refining UPD and then further refined to a form that is transformable by $\mathcal{S}$ using standard refinement rules. Further details are omitted.

Although the above deletion algorithm does result in an ordered tree, it may result in a tree that is very unbalanced. A better solution uses the following definition of $join$:

$$\begin{array}{rcl} join(L)(R) & = & \textbf{if}\ L = \epsilon \rightarrow R \\ & & [\!]\ \ L \neq \epsilon \wedge R = \epsilon \rightarrow L \end{array}$$

$$
\begin{array}{rcl}
 & & \;[]\;\; L \neq \epsilon \wedge R \neq \epsilon \rightarrow \\
 & & \quad \mathsf{node}(\mathit{least}(R), L, \mathit{remove\_least}(R))\ \mathbf{fi} \\
\\
\mathit{least}(\mathsf{node}(a, L, R)) & = & \mathbf{if}\ L = \epsilon \rightarrow a \\
 & & \;[]\;\; L \neq \epsilon \rightarrow \mathit{least}(L)\ \mathbf{fi} \\
\\
\mathit{remove\_least}(\mathsf{node}(a, L, R)) & = & \mathbf{if}\ L = \epsilon \rightarrow R \\
 & & \;[]\;\; L \neq \epsilon \rightarrow \mathsf{node}(a,\ \mathit{remove\_least}(L),\ R)\ \mathbf{fi}.
\end{array}
$$

Here the auxiliary function *least* is a special case of *SRCH* (see Section 4.3) while the function *remove_least* is a special case of *UPD*. In both cases, the guards are identical so that applying the *SRCH* refinement template to *least* and the *UPD* refinement template to *remove_least* would result in identical copies of the same search loop. Using standard refinement rules, these may be merged into a single loop thereby avoiding the same search being performed twice. Alternatively, we could invent a new refinement template for a function that is a merge of *SRCH* and *UPD* returning a value-tree pair:

$$
\begin{array}{rcl}
\mathit{UPD2}(\epsilon) & = & (E1, F1) \\
\mathit{UPD2}(\mathsf{node}(b, L, R)) & = & \mathbf{if}\ \mathit{CT}(\mathsf{node}(b, L, R)) \rightarrow \\
 & & \qquad (E2(\mathsf{node}(b, L, R)),\ F2(\mathsf{node}(b, L, R))) \\
 & & \;[]\;\; \mathit{CL}(\mathsf{node}(b, L, R)) \rightarrow \\
 & & \qquad (e,\ \mathsf{node}(b, L', R))\ \mathbf{where}\ (e, L') = \mathit{UPD2}(L) \\
 & & \;[]\;\; \mathit{CR}(\mathsf{node}(b, L, R)) \rightarrow \\
 & & \qquad (e,\ \mathsf{node}(b, L, R'))\ \mathbf{where}\ (e, R') = \mathit{UPD2}(R) \\
 & & \mathbf{fi}.
\end{array}
$$

The statement $\{\mathit{Is\_Whole}(t)\}\ x, t := \mathit{UPD2}(t)$ has an iterative refinement similar to the refinement of $\{\mathit{Is\_Whole}(t)\}\ t := \mathit{UPD}(t)$.

7.2 Correctness

The correctness of the refinement rule for *UPD* is checked using the standard loop-verification rule. To define an invariant, the following auxiliary function is introduced ($E1$, $E2$, CT, CL, and CR are the same as in the definition of *UPD*):

$$
\begin{array}{rcl}
\mathit{UPD}'(t, m) & = & \mathbf{if}\ t/m = \epsilon \rightarrow t[m \backslash E1] \\
 & & \;[]\;\; t/m \neq \epsilon \wedge \mathit{CT}(t/m) \rightarrow t[m \backslash E2(t/m)] \\
 & & \;[]\;\; t/m \neq \epsilon \wedge \mathit{CL}(t/m) \rightarrow \mathit{UPD}'(t, m\langle \mathsf{left} \rangle) \\
 & & \;[]\;\; t/m \neq \epsilon \wedge \mathit{CR}(t/m) \rightarrow \mathit{UPD}'(t, m\langle \mathsf{right} \rangle) \\
 & & \mathbf{fi}.
\end{array}
$$

Now we have the following lemma:

$$\mathit{UPD}'(t, \langle\rangle) \ = \ \mathit{UPD}(t).$$

The proof of this is outlined in Appendix A. As a loop invariant for verifying the loop, we use:

$$m \in \mathit{paths}(t) \ \wedge \ \mathit{UPD}'(t, m) \ = \ \mathit{UPD}(t),$$

while $\mathit{size}(t/m)$ is used as a variant.

8 Soundness

In this section, we outline how the transformations presented in Sections 3 and 6 are shown to be sound, that is, any specification resulting from a transformation is a correct data refinement of the source specification.

We look at data refinement in general, then develop the abstraction relation required for the data refinement and apply it to the pointer-introduction transformation.

8.1 Data Refinement Overview

Data refinement involves replacing abstract program variables with concrete program variables preserving an abstraction relation between them. Let S be a statement with program variables u, a, and let T be a statement with variables u, c (a represents the abstract variables that are replaced by the concrete variables c while u represents variables that are common to both the abstract and concrete statements). S is data-refined by T under abstraction relation R, written $S \sqsubseteq_R T$, if the following holds, for all postconditions q where c **nfi** q [1, 2, 15, 17] (x **nfi** P means that x is not free in P):

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

We also make 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 $\mathcal{D}_R^{a,c}[\![\ S\ ]\!]$, that is:

$$S \sqsubseteq_R \mathcal{D}_R^{a,c}[\![\ S\ ]\!] \quad and \quad S \sqsubseteq_R T \ \Rightarrow\ \mathcal{D}_R^{a,c}[\![\ S\ ]\!] \sqsubseteq T.$$

We calculate a data refinement of a statement S under R by calculating a refinement of $\mathcal{D}_R^{a,c}[\![\ S\ ]\!]$. Rules that support such calculation by distributing $\mathcal{D}_R^{a,c}$ through the structure of S may be found in [9, 15, 17, 18]. Some of these rules are repeated in Figure 9. The first rule shows that $\mathcal{D}_R^{a,c}$ distributes through sequential composition. The second and third rules show that $\mathcal{D}_R^{a,c}$ distributes through if-statements and loops provided the guards are equivalent under the abstraction relation. The fourth and fifth rules show the conditions under which generalised assignments and basic assignments may be data refined.

8.2 Basic Statement Notation

To show that the transformation function $\mathcal{S}$ is sound, we need to show that for any transformable statement S, $\mathcal{S}[\![\ S\ ]\!]$ is a data refinement of S under some abstraction relation. We will show this for a more basic statement language for which statements used up to now are syntactic sugar. This more basic language differs only in the way it treats procedures.

We declare a (parameterless) procedure *Proc*, by declaring **procedure** $Proc \mathrel{\hat{=}} S$, where S is some statement. The effect of a call to *Proc* is the same as substituting program statement S for *Proc*. The following substitutions are used as parameter-passing mechanisms:

- *Substitution by Value:* In the statement $S[\mathbf{val}\ x \backslash E]$, x is assumed to be some constant in S (and hence cannot be assigned to) and $S[\mathbf{val}\ x \backslash E]$ simply substitutes E for x, e.g., $(x := x + y)[\mathbf{val}\ y \backslash 7] \ = \ x := x + 7$.
- *Substitution by Reference:* In the statement $S[\mathbf{ref}\ x \backslash y]$, all occurrences of program variable x are replaced by y, e.g., $(x := x+z)[\mathbf{ref}\ x \backslash y] \ = \ y := y+z$. Recall from Section 3.2 that, for simplicity, parameter aliasing (i.e., replacing more than one reference variable with the same actual variable) is not allowed.

$$\mathcal{D}_R^{a,c}[\![\ S_1;\ S_2\ ]\!] \ \sqsubseteq\ \mathcal{D}_R^{a,c}[\![\ S_1\ ]\!];\ \mathcal{D}_R^{a,c}[\![\ S_2\ ]\!]$$

$$\frac{p \wedge R \ \Rightarrow\ G = H}{\mathcal{D}_R^{a,c}[\![\ \{p\}\ \mathbf{do}\ G \rightarrow S\ \mathbf{od}\ ]\!] \ \sqsubseteq\ \mathbf{do}\ H \rightarrow \mathcal{D}_R^{a,c}[\![\ S\ ]\!]\ \mathbf{od}}$$

$$\frac{p \wedge R \ \Rightarrow\ G_i = H_i, \text{each } i}{\begin{array}{l}\mathcal{D}_R^{a,c}[\![\ \{p\}\ \mathbf{if}\ G_1 \rightarrow S_1\ [\!]\ \cdots\ [\!]\ G_n \rightarrow S_n\ \mathbf{fi}\ ]\!] \\ \sqsubseteq\ \mathbf{if}\ H_1 \rightarrow \mathcal{D}_R^{a,c}[\![\ S_1\ ]\!]\ [\!]\ \cdots\ [\!]\ H_n \rightarrow \mathcal{D}_R^{a,c}[\![\ S_n\ ]\!]\ \mathbf{fi}\end{array}}$$

$$\frac{p \wedge R \wedge Q \ \Rightarrow\ (\exists a' \cdot R[a, c, u\backslash a', c', u'] \wedge P)}{\mathcal{D}_R^{a,c}[\![\ \{p\}\ a, u := a', u' \mid P\ ]\!] \ \sqsubseteq\ c, u := c', u' \mid Q}$$

$$\frac{p \wedge R \ \Rightarrow\ E = F}{\mathcal{D}_R^{a,c}[\![\ \{p\}\ u := E\ ]\!] \ \sqsubseteq\ u := F}$$

Figure 9: Data Refinement Laws

Parameters are added to a procedure by declaring, for example,

$$\mathbf{procedure}\ \mathit{Proc}(\mathbf{val}\ x) \mathrel{\hat{=}} S,$$

and the call $\mathit{Proc}(E)$ is then short for $S[\mathbf{val}\ x\backslash E]$.

A recursive statement is defined as a least fixed point (least with respect to the refinement order), written $\mu X \cdot S[X]$. The declaration **procedure** $\mathit{Proc} \mathrel{\hat{=}} S[\mathit{Proc}]$ is short for **procedure** $\mathit{Proc} \mathrel{\hat{=}} \mu X \cdot S[X]$, and, for example, **procedure** $\mathit{Proc}(\mathbf{val}\ x) \mathrel{\hat{=}} S[\mathit{Proc}(E)]$ is short for **procedure** $\mathit{Proc} \mathrel{\hat{=}} \mu X \cdot S[X[\mathbf{val}\ x\backslash E]]$. A set of mutually recursive procedure definitions is of the form:

$$\mathbf{procedure}\ p_1 \mathrel{\hat{=}} S_1 \quad \cdots \quad \mathbf{procedure}\ p_n \mathrel{\hat{=}} S_n,$$

where each S_i may contain recursive calls to any of $p_1 \ldots p_n$. These define a single n-tuple of statements:

$$\mu X \cdot (S_1', \ldots, S_n'),$$

where S_i' is S_i with calls to any p_j replaced by a call to X_j, the j-th component of X. Procedure p_i is then the j-th component of this tuple, written $(\mu X \cdot (S_1', \ldots, S_n'))_j$.

The syntax of the basic statement language is given in Figure 10. Data refinement laws for local variables and procedure calling are given in Appendix B.

To deal with potentially undefined values, e.g., $\mathit{root}(t)$ when $t = \epsilon$ or $t = *$, we assume that assignments $x := E$ are short for

$$\{\mathit{def}(E)\}\ x := E.$$

Here, $\mathit{def}(E)$ is a predicate describing when an expression E is well defined, e.g.,

$$\begin{array}{rcl}\mathit{def}(\mathit{root}(t)) &=& \mathit{Is\_Node}(t) \\ \mathit{def}(t \neq \epsilon \wedge \mathit{root}(t) = a) &=& t \neq *.\end{array}$$

The predicate $\mathit{def}(E)$ may be defined over the structure of E. A refinement of $\{\mathit{def}(E)\}\ x := E$ may do anything (including abort) in states where $\mathit{def}(E)$ does not hold. We assume that procedure calls are preceded by assertions stating that **val** parameters are well defined and that if-statements and loops are preceded by assertions stating that their guards are well defined. We also assume that loop bodies end with a similar assertion.

$$
\begin{array}{lcl}
\mathit{Stmt} & ::= & x_1, \ldots, x_n := E_1, \ldots, E_n \\
& | & \mathit{Stmt};\ \mathit{Stmt} \\
& | & \{\mathit{Pred}\} \\
& | & \mathbf{do}\ \mathit{Pred} \rightarrow \mathit{Stmt}\ \mathbf{od} \\
& | & \mathbf{if}\ \mathit{Pred} \rightarrow \mathit{Stmt}\ [\!]\ \ \cdots\ [\!]\ \mathit{Pred} \rightarrow \mathit{Stmt}\ \mathbf{fi} \\
& | & (\mu X \cdot (\mathit{Stmt}, \ldots, \mathit{Stmt}))_i \\
& | & \mathit{Stmt}[\mathbf{val}\ x \backslash E] \\
& | & \mathit{Stmt}[\mathbf{ref}\ x \backslash y] \\
& | & |[\mathbf{var}\ x \bullet \mathit{Stmt}]|.
\end{array}
$$

Figure 10: Syntax for basic statements.

8.3 Data Refinement of Pointer Structures

Rather than requiring that different abstract trees be represented by disjoint graphs in the store to avoid side effects, we only require that they be disjoint modulo $*$. The set of significant pointers in the representation of tree t starting at pointer p in store σ is given by $\mathit{points}(t, p, \sigma)$ which is defined as follows:

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

Here any pointers representing $*$ are ignored. Note that $\mathit{points}(t, p, \sigma)$ is well defined when $\mathit{Match}(t, p, \sigma)$ holds. Now the representations of trees t_i, t_j starting at p_i, p_j respectively are disjoint modulo $*$ iff $\mathit{Disjoint}(t_i, p_i, t_j, p_j, \sigma)$:

$$
\mathit{Disjoint}(t_i, p_i, t_j, p_j, \sigma) \quad \hat{=} \quad \mathit{points}(t_i, p_i, \sigma)\ \cap\ \mathit{points}(t_j, p_j, \sigma)\ =\ \{\}.
$$

The graph of the interconnections between significant nodes representing tree t starting at p in σ is given by $\mathit{graph}(t, p, \sigma)$ which is defined as follows:

$$
\begin{array}{lllcl}
\mathit{graph}(\ \epsilon, & p, & \sigma\ ) & \hat{=} & \{\} \\
\mathit{graph}(\ *, & p, & \sigma\ ) & \hat{=} & \{\} \\
\mathit{graph}(\ \mathsf{node}(b, L, R), & p, & \sigma\ ) & \hat{=} & \{\ (p, q) \mid q = \sigma(p).\mathit{left}\ \wedge\ \mathit{Is\_Node}(L)\ \}\ \cup \\
 & & & & \{\ (p, q) \mid q = \sigma(p).\mathit{right}\ \wedge\ \mathit{Is\_Node}(R)\ \}\ \cup \\
 & & & & \mathit{graph}(L,\ \sigma(p).\mathit{left},\ \sigma)\ \cup \\
 & & & & \mathit{graph}(R,\ \sigma(p).\mathit{right},\ \sigma).
\end{array}
$$

Again $\mathit{graph}(t, p, \sigma)$ is well defined when $\mathit{Match}(t, p, \sigma)$ holds. This definition excludes arcs whose target represents an empty tree or a $*$-tree.

We shall require that such graphs be acyclic. If t does not contain $*$, and $\mathit{Match}(t, p, \sigma)$ holds, then $\mathit{graph}(t, p, \sigma)$ 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 graph, e.g., consider $t_1 = \mathsf{node}(b, \mathsf{node}(b, *, \epsilon), \epsilon)$ and $\sigma = \{\ (p_1,\ \mathit{mk\_node}(b, p_1, \mathsf{nil}))\ \}$. We will need to state explicitly that $\mathit{graph}(t, p, \sigma)$ be acyclic:

$$
\mathit{Acyclic}(t, p, \sigma) \quad \hat{=} \quad \mathit{graph}(t, p, \sigma)^+\ \cap\ \mathit{ID}_{\mathit{TreePtr}} = \{\}.
$$

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

We are not excluding the possibility of having cyclic structures at the pointer level. To see why this is, consider the following two assignments:

$$
t_2, t_1\ :=\ t_1, *\ ;\ \ t_2, t_1\ :=\ t_2[\mathsf{left} \backslash t_1], *.
$$

These are transformed by $\mathcal{S}$ to

$$p_2 := p_1 \ ; \ \ p_2\hat{}.left := p_1,$$

which is equivalent to

$$p_2 := p_1 \ ; \ \ p_2\hat{}.left := p_2.$$

This clearly results in a cyclic pointer structure. However, in the resulting σ, $graph(t_2, p_2, \sigma)$ is not cyclic. For example, if $t_1 = \mathsf{node}(a, \epsilon, \epsilon)$ initially, then executing the above assignments will result in $t_2 = \mathsf{node}(a, *, \epsilon)$ and, according to the above definition, $graph(t_2, p_2, \sigma) = \{\}$.

It may appear that cycles, such as arise in the above example, are dangerous in that they could introduce non-termination. Fortunately this is not the case. This is because cycles such as these can only arise from trees containing $*$ and, since we do not (indeed cannot) provide a transformation for the test $t = *$, an algorithm that relies on the test $t = *$ is not transformable by $\mathcal{S}$.

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

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

Note that $Is\_Whole(t) \wedge Match(t, p, \sigma) \ \Rightarrow \ get\_tree(p, \sigma) = t$.

As stated earlier, rather than representing a path m_i directly, we represent the subtree t_i/m_i with a pointer q_i. As part of the abstraction relation, we require the function *follow*, which returns the pointer reached by following path m_i starting at pointer p_i:

$$\begin{array}{llll} follow(\ p, & \langle\rangle, & \sigma \ ) & \hat{=} \ p \\ follow(\ p, & \langle \mathsf{left} \rangle m, & \sigma \ ) & \hat{=} \ follow(\sigma(p).left, m, \sigma) \\ follow(\ p, & \langle \mathsf{right} \rangle m, & \sigma \ ) & \hat{=} \ follow(\sigma(p).right, m, \sigma). \end{array}$$

Note that $follow(p, m, \sigma)$ is well defined when $m \in paths(t) \ \wedge \ Match(t, p, \sigma)$ holds.

The abstraction relation associated with the representation of tree t and path m is then defined as follows: for $t : Tree$, $m : Path$, $p, q : TreePtr$, $\sigma : TreeStore$,

$$\begin{array}{l} AbsTreePath(t, m, p, q, \sigma) \ \hat{=} \\ \quad AbsTree(t, p, \sigma) \ \wedge \\ \quad m \in paths(t) \ \wedge \\ \quad q \ = \ follow(p, m, \sigma). \end{array}$$

The abstraction relation associated with the representation of tree t and a pasting path $\overline{m}$ is defined as follows: for $t : Tree$, $\overline{m} : Path$, $p, q, r : TreePtr$, $\sigma : TreeStore$,

$$\begin{array}{l} AbsTreePastePath(t, \overline{m}, p, q, r, \sigma) \ \hat{=} \\ \quad AbsTreePath(t, \overline{m}, p, q, \sigma) \ \wedge \\ \quad (\overline{m} \neq \langle\rangle \ \Rightarrow \ r = follow(p, front(\overline{m}), \sigma)). \end{array}$$

In general, the pointer-introduction transformation involves replacing t_I, m_J, $\overline{m}_K$, where $J \subseteq I$, $K \subseteq I$ and $J \cap K = \{\}$, with pointers p_I, q_J, q_K, r_K and store σ. The abstraction relation used for this replacement is $INV[I, J, K]$, where

$$\begin{array}{lll} INV[I, J, K] & \hat{=} & (\forall i \in I \cdot AbsTree(t_i, p_i, \sigma)) \ \wedge \\ & & (\forall i \in J \cdot AbsTreePath(t_i, m_i, p_i, q_i, \sigma)) \ \wedge \\ & & (\forall i \in K \cdot AbsTreePastePath(t_i, \overline{m}_i, p_i, q_i, r_i, \sigma)) \ \wedge \\ & & (\forall i, j \in I | i \neq j \cdot Disjoint(t_i, p_i, t_j, p_j, \sigma)). \end{array}$$

We add more precision to the definition of $\mathcal{S}[\![S]\!]$ by making explicit those tree and path variables that are in scope in S: if S operates on t_I, m_J, $\overline{m}_K$, then the transformed statement is calculated with $\mathcal{S}_{I,J,K}[\![S]\!]$. For the most part, $\mathcal{S}_{I,J,K}[\![S]\!]$ is the same as defined in Sections 3 and 6 with I, J, K simply being carried around, e.g.,

$$\mathcal{S}_{I,J,K}[\![S_1;\ S_2]\!] \ \hat{=}\ \mathcal{S}_{I,J,K}[\![S_1]\!]\ ;\ \mathcal{S}_{I,J,K}[\![S_2]\!].$$

This subscripting is also applied to $\mathcal{E}[\![E]\!]$ and $\mathcal{E}[\![P]\!]$. Now, any transformation rule that mentions a tree t_i is only applicable if $i \in I$ (similarly for paths m_j, $\overline{m}_k$).

The only significant additions to $\mathcal{S}_{I,J,K}[\![S]\!]$ are to do with local variables and parameters:

$$\begin{array}{rcl}
\mathcal{S}_{I,J,K}[\![\ |[\mathbf{var}\ t_i \bullet S\ ]|\ ]\!] & \hat{=} & |[\mathbf{var}\ p_i \bullet \mathcal{S}_{I\cup\{i\},J,K}[\![S]\!]\ ]| \\
\mathcal{S}_{I,J,K}[\![\ |[\mathbf{var}\ m_i \bullet S\ ]|\ ]\!] & \hat{=} & |[\mathbf{var}\ q_i \bullet \mathcal{S}_{I,J\cup\{i\},K}[\![S]\!]\ ]| \\
\mathcal{S}_{I,J,K}[\![\ |[\mathbf{var}\ \overline{m}_i \bullet S\ ]|\ ]\!] & \hat{=} & |[\mathbf{var}\ q_i, r_i \bullet \mathcal{S}_{I,J,K\cup\{i\}}[\![S]\!]\ ]|.
\end{array}$$

In each of the following, let $I' = (I \setminus \{j\}) \cup \{i\}$, $J' = (J \setminus \{j\}) \cup \{i\}$, $K' = (K \setminus \{j\}) \cup \{i\}$.

$$\mathcal{S}_{I,J,K}[\![S[\mathbf{val}\ t_i \backslash E[t_j]]]\!] \ \hat{=}\ \mathcal{S}_{I',J,K}[\![S]\!][\mathbf{val}\ p_i \backslash \mathcal{V}[\![E[t_j]]\!]]$$

$$\mathcal{S}_{I,J,K}[\![S[\mathbf{ref}\ t_i \backslash t_j]]\!] \ \hat{=}\ \mathcal{S}_{I',J,K}[\![S]\!][\mathbf{ref}\ p_i \backslash p_j]$$

$$\begin{array}{l}\mathcal{S}_{I,J,K}[\![S[\mathbf{ref}\ t_i \backslash t_j][\mathbf{val}\ m_i \backslash E[t_j]]]\!] \ \hat{=} \\ \qquad \mathcal{S}_{I',J',K}[\![S]\!][\mathbf{ref}\ p_i \backslash p_j][\mathbf{val}\ q_i \backslash \mathcal{V}[\![E[t_j]]\!]]\end{array}$$

$$\begin{array}{l}\mathcal{S}_{I,J,K}[\![S[\mathbf{ref}\ t_i \backslash t_j][\mathbf{ref}\ m_i \backslash m_j]]\!] \ \hat{=} \\ \qquad \mathcal{S}_{I',J',K}[\![S]\!][\mathbf{ref}\ p_i \backslash p_j][\mathbf{ref}\ q_i \backslash q_j]\end{array}$$

$$\begin{array}{l}\mathcal{S}_{I,J,K}[\![S[\mathbf{ref}\ t_i \backslash t_j][\mathbf{ref}\ \overline{m}_i \backslash \overline{m}_j]]\!] \ \hat{=} \\ \qquad \mathcal{S}_{I',J,K'}[\![S]\!][\mathbf{ref}\ p_i \backslash p_j][\mathbf{ref}\ q_i \backslash q_j][\mathbf{ref}\ r_i \backslash r_j].\end{array}$$

The first two rules above are applicable when t_j has no path variables in scope. Each of the parameter cases permitted by the restrictions of Sections 3 and 6 are covered by the above definitions.

Soundness of the pointer-introduction rules is given by the following theorem which states that statement S is data-refined by its transformation $\mathcal{S}_{I,J,K}[\![S]\!]$, where I, J and K include the indices of tree and path variables that are global in statement S as well as other tree and path variables that are not global. It is important to include these extra variables to allow S to be used in a larger context which includes other tree and path variables (for example, in procedure calling). Although the extra variables are independent at the abstract level, the concrete implementations share the store with the context.

Theorem 1 *Let S be a statement that is transformable by the rules of Sections 3 and 6 and has been converted to basic form. Let I_1, J_1, K_1 respectively be the set of indices of tree variables, path variables and pasting path variables that are global in S (with $J_1, K_1 \subseteq I_1$ and $J_1 \cap K_1 = \{\}$). Let I_2, J_2, K_2 be any sets of indices of tree, path and pasting path variables that are disjoint from I_1, J_1, K_1 respectively and also disjoint from the indices of any local variables in S (with $J_2, K_2 \subseteq I_2$ and $J_2 \cap K_2 = \{\}$).*

Let $I = I_1 \cup I_2$, $J = J_1 \cup J_2$, $K = K_1 \cup K_2$ and

$$\begin{array}{rcl}
a & = & t_I \ \cup\ m_J \ \cup\ \overline{m}_K \\
c & = & p_I \ \cup\ q_J \ \cup\ q_K \ \cup\ r_K \ \cup\ \{\sigma\} \\
R & = & INV[I,J,K].
\end{array}$$

Then

$$\mathcal{D}_R^{a,c}[\![\ S\ ]\!] \ \sqsubseteq\ \mathcal{S}_{I,J,K}[\![\ S\ ]\!].$$

□

Proof outline: Proof of this theorem is by structural induction over S. We deal with the inductive cases first. These follow easily from the distribution rules for data refinement. The distribution rules for loops and if-statements require:

$$INV[I,J,K] \ \wedge\ def(E) \quad\Rightarrow\quad E = \mathcal{E}[\![\ E\ ]\!]. \tag{6}$$

Implication (6) follows easily by structural induction on E: in the case of root expressions, we have, for example, that

$$INV[I,J,K] \ \wedge\ Is\_Node(t_i) \qquad\Rightarrow\qquad root(t_i) \ =\ p_i\hat{}.root,$$

while in the case of simple checks for equality, we have, for example, that

$$INV[I,J,K] \qquad\Rightarrow\qquad (t_i = \epsilon) \ =\ (p_i = \mathsf{nil}).$$

The other inductive case that introduces proof obligations is for **val** parameters in procedure calls (see Appendix B). This case requires, for example, that

$$INV[I,J,K] \ \wedge\ def(E) \ \wedge\ pre \qquad\Rightarrow\qquad INV[I,J,K][t_i, p_i \backslash E, \mathcal{V}[\![\ E\ ]\!]].$$

Here, *pre* is the list of preconditions on **val** parameters preceding a procedure call (see Section 3.2). Again this follows easily from the definition of $\mathcal{V}$, e.g., for the case of $\mathcal{V}[\![\ left(t_i)\ ]\!]$, we have to show that

$$INV[I,J,K] \quad\Rightarrow\quad INV[I,J,K][t_i, p_i \backslash left(t_i), \sigma(p_i).left].$$

This follows from

$$Is\_Node(t) \wedge AbsTree(t,p,\sigma) \ \Rightarrow\ AbsTree(left(t), \sigma(p).left, \sigma).$$

The base cases of the inductive proof on statements are the assignments. Assignments to program variables other than tree or path variables are taken care of by (6) and the data-refinement rules for assignment in Figure 9.

The remaining base cases are the assignments to trees and paths. Each of these is also taken care of by the data-refinement rules for assignment in Figure 9. For example, to show that

$$\mathcal{D}[\![\ \{Is\_Node(t_i/m_i)\} m_i := m_i\langle \mathsf{left}\rangle\ ]\!] \quad\sqsubseteq\quad q_i := q_i\hat{}.left,$$

we need to show that

$$Is\_Node(t_i/m_i) \ \wedge\ INV[I,J,K] \qquad\Rightarrow\qquad INV[I,J,K][m_i, q_i \backslash m_i\langle\mathsf{left}\rangle, \sigma(q_i).left].$$

This follows from

$$\begin{aligned} & Is\_Node(t/m) \wedge AbsTreePath(t,m,p,q,\sigma) \\ \Rightarrow\quad & AbsTreePath(t, m\langle\mathsf{left}\rangle, p, \sigma(q).left, \sigma), \end{aligned}$$

which follows from

$$\begin{aligned} & Is\_Node(t/m) \ \wedge\ Match(t,p,\sigma) \ \wedge\ m \in paths(t) \ \wedge\ q = follow(p,m,\sigma) \\ & \quad\Rightarrow\quad \sigma(q).left = follow(p, m\langle\mathsf{left}\rangle, \sigma). \end{aligned}$$

Before concluding this proof outline, we comment on the requirement that the pointer representations be acyclic and that separate trees have disjoint representations (modulo $*$). Disjointness is clearly required to prevent an assignment to one tree from having a side effect on another. The requirement for acyclicity is necessary in order to ensure that the cut operation preserves disjointness. For example, without the acyclicity requirement, the tree $t_1 = \mathsf{node}(b, \mathsf{node}(b, *, \epsilon), \epsilon)$ could be represented by the acyclic store $\sigma = \{ (p_1,\ mk\_node(b, p_1, \mathsf{nil})) \}$. Now, the cut operation $t_1, t_2 := t_1 \upharpoonright \mathsf{left}, t_1/\mathsf{left}$ would be implemented by $p_2 := p_1$ which would clearly violate disjointness. Also, disjointness of t_i and t_j is necessary to ensure that paste preserves acyclicity (modulo $*$).

9 Conclusions

We have described a function $\mathcal{S}$ that syntactically transforms algorithms on enriched trees into efficient pointer implementations. We outlined a proof showing that the target code generated by $\mathcal{S}$ is always a correct data refinement of the source code. The use of such a transformation function was partly inspired by the distribution rules for data refinement of Figure 9 and Appendix B.

The transformation function $\mathcal{S}$ is crucially dependent on having identified some commonly-used pointer manipulations on tree-based structures and providing representations of these manipulations in terms of abstract trees. We required two simple enrichments of abstract trees: paths, which allow for access of subtrees without using recursive descent, and the $*$-tree, which provides an elegant way of dealing with sharing that arises during manipulation of structures. Use of the $*$-tree, along with the various syntactic restrictions on what is transformable, allow us to avoid aliasing. This simplifies the reasoning considerably and allows the pointer-introduction transformations to work smoothly. As well as traversal and manipulation of trees, we are able to deal safely with storage allocation and deallocation (though we have ignored the potential problem of running out of memory).

Several approaches to dealing with correctness of pointer algorithms exist [3, 10, 11]. These provide general rules for reasoning about programs at the level of pointers. Their use can be quite cumbersome as they usually involve complex assertions.

Möller introduced a transformational approach to pointer algorithms in [12, 13]. He explicitly treats the store as a partial map and provides algebraic laws for partial maps that can be used to transform operations on abstract lists and trees to pointer algorithms. Möller's work served as a strong basis for our work on the refinement of trees by pointers. From experience, we found the derivation of iterative algorithms on pointers using Möller's approach to be quite complicated. By identifying abstractions of common operations, providing correct refinements of these and then putting these together with the rules for distributing data refinement, we can shield many of these complications.

The idea of using paths as abstractions for pointers is not new either. Cartwright et al proposed extending Lisp-like data structures with paths in order to replace pointers [6]. They provide some operations similar to ours such as extending a path to the left or to the right. However, they don't provide a cut operation or deal with sharing in the way that we do.

Many operators on trees are most naturally described as applicative-style recursive functions. In addition to the transformation function $\mathcal{S}$, we provide some rules that can be used to derive imperative algorithms on enriched trees from such recursive functions. Essentially we provide a set of refinement templates for several classes of recursive functions on trees.

In this paper, we imposed the restriction that each tree may have at most one path. However, [4] does support multiple paths. The difficulty of having multiple paths is that a paste operation can destroy the abstraction relation between a tree and other paths if the path used to perform the paste is a prefix of some other path. In [4], this is dealt with by adding the following assertion to the abstract paste operation:

$$\{\ \neg(\exists k \mid k \neq j \cdot m_i^j \preceq m_i^k)\ \}.$$

Here, m_i^j is the path used to perform the paste to t_i and $m \preceq m'$ means that m is a prefix of m'. We avoided multiple paths in this paper to simplify the description of the transformation function $\mathcal{S}$ and its soundness proof. Furthermore, a single path is sufficient for many algorithms involving trees including insertion and deletion on balanced trees.

The function $\mathcal{S}$ is presented as a set of straightforward syntactic transformations of statements, predicates and expressions. The restrictions on what is transformable are also syntactic. Therefore it should be possible to embed these transformations in a tool that supports program refinement such as [5]. It should also be possible to provide support for the applicative-to-imperative refinement schemas in such a tool.

Although we have restricted ourselves to binary trees, the approach described here should extend easily to n-ary trees and more general recursive data structures (such as abstract syntax trees for languages). We could also consider enriching the concrete pointer structures, for example, by having doubly-linked structures. In either case, we could then extend the set of abstract operations for which pointer refinements are provided (e.g., retracting a path in the case of doubly-linked pointers) and also extend the set of applicative-to-imperative refinement schemas.

Acknowledgements

I am grateful to the anonymous SCP referees for detailed constructive comments and for encouraging me to improve earlier work. Several members of the DSSE group at Southampton also provided useful feedback.

A Auxiliary Update Function

Lemma 1

$$\begin{array}{lcl} m \in \mathit{paths}(L) & \Rightarrow & \mathit{UPD}'(\mathsf{node}(a, L, R), \langle\mathsf{left}\rangle m) \;=\; \mathsf{node}(a, \mathit{UPD}'(L, m), R) \\ m \in \mathit{paths}(R) & \Rightarrow & \mathit{UPD}'(\mathsf{node}(a, L, R), \langle\mathsf{right}\rangle m) \;=\; \mathsf{node}(a, L, \mathit{UPD}'(R, m)). \end{array}$$

This lemma is proven using standard fixed-point reasoning treating UPD' as a relation defined by a least (w.r.t. relational inclusion) fixed-point.

Lemma 2

$$m \in \mathit{paths}(t) \;\;\Rightarrow\;\; \mathit{UPD}'(t, m) \;=\; t[m \backslash \mathit{UPD}(t/m)].$$

This lemma is proven using structural induction over t and m and Lemma 1.

Corollary 1

$$\mathit{UPD}'(t, \langle\rangle) \;=\; \mathit{UPD}(t).$$

B Additional Data Refinement Laws

$$\frac{\begin{array}{c} a_1, c_1 \ \mathbf{nfi} \ R \\ R \Rightarrow (\exists a_1, c_1 \cdot Q) \\ S \text{ independent of initial value of } a_1 \end{array}}{\mathcal{D}_R^{a_0,c_0}[\![\ |[\mathbf{var} \ a_1 \bullet S]| \ ]\!] \ \sqsubseteq \ |[\mathbf{var} \ c_1 \bullet \mathcal{D}_{R \wedge Q}^{(a_0,a_1),(c_0,c_1)}[\![\ S \ ]\!] \ ]|}$$

$$\frac{p \wedge R \quad \Rightarrow \quad R[a_0, c_0, u_0 \backslash E, G, F] \ \wedge \ F = H}{\begin{array}{l} \mathcal{D}_R^{(a_0,a_1),(c_0,c_1)}[\![\ \{p\} \ S[\mathbf{val} \ a_0, u_0 \backslash E, F] \ ]\!] \\ \sqsubseteq \ \ \mathcal{D}_R^{(a_0,a_1),(c_0,c_1)}[\![\ S \ ]\!][\mathbf{val} \ c_0, u_0 \backslash G, H] \end{array}}$$

$$\frac{p \wedge R \quad \Rightarrow \quad Q[a_2, c_2, u_2 \backslash E, G, F] \ \wedge \ F = H}{\begin{array}{l} \mathcal{D}_R^{(a_0,a_1),(c_0,c_1)}[\![\ \{p\} \ S[\mathbf{val} \ a_2, u_2 \backslash E, F] \ ]\!] \\ \sqsubseteq \ \ \mathcal{D}_{R \wedge Q}^{(a_0,a_1,a_2),(c_0,c_1,c_2)}[\![\ S \ ]\!][\mathbf{val} \ c_2, u_2 \backslash G, H] \end{array}}$$

$$\begin{array}{l} \mathcal{D}_R^{(a_0,a_1),(c_0,c_1)}[\![\ S[\mathbf{ref} \ a_2, u_2 \backslash a_0, u_0] \ ]\!] \\ \sqsubseteq \ \ \mathcal{D}_{R[a_0,c_0,u_0 \backslash a_2,c_2,u_2]}^{(a_1,a_2),(c_1,c_2)}[\![\ S \ ]\!][\mathbf{ref} \ c_2, u_2 \backslash c_0, u_0] \end{array}$$

References

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

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

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

[4] M.J. Butler. Calculational derivation of algorithms on tree-based pointer structures. In He Jifeng, J. Cooke, and P. Wallis, editors, *BCS-FACS Refinement Workshop*, Workshops in Computing. Springer–Verlag, 1996.

[5] M.J. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The refinement calculator: proof support for program refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific'97*, Springer Series in Discrete Mathematics and Theoretical Computer Science, 1997.

[6] R. Cartwright, R. Hood, and P. Matthews. Paths: An abstract alternative to pointers. In *Proceedings 8th ACM Symposium on Principles of Programming Languages*, pages 14–27. ACM Press, 1981.

[7] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier Science Publishers, 1990.

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

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

[10] 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.

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

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

[13] B. Möller. Calculating with pointer structures. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi, IFIP TC2/WG2.1 Working Conference*, pages 24–48. Chapman & Hall, 1997.

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

[15] C.C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Formal Approaches to Computing and Information Technology. Springer, 1994.

[16] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):298–306, 1987.

[17] J.M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

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