Quicksort via Bird's Tree Fusion Transformation*

Tjark Weber[1] and James Caldwell[2]

[1] Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
`tjark.weber@gmx.de`
[2] Department of Computer Science, University of Wyoming
Laramie, Wyoming, USA, 82071-3315
`jlc@cs.uwyo.edu`

Abstract. In this paper we present a Nuprl formalization and proof of Bird's fusion theorem for trees. We apply the theorem to a derivation of quicksort.

1 Introduction

Many algorithms can be specified as the composition of a function that constructs an intermediate data structure from the given input, and another function that traverses the intermediate data structure to extract the requested information.

Bird's fusion theorem [1] proves that if the first function is an anamorphism and the second function is a catamorphism, these two functions can be combined into a single function, thereby eliminating the intermediate data structure constructed by the anamorphism.

This paper presents a formalization of the fusion theorem for the special case where the underlying data structure is the type of binary trees and then applied the theorem to the derivation of the quicksort algorithm. The formalization presented here is partially based on a formalization of Bird's fusion transformation in PVS by N. Shankar [7].

2 Binary Trees

A *binary tree* (over some type T) is a type of data structure in which each element is attached to zero or two elements directly beneath it. We use the following inductive definition after [2].

Definition 1 (Binary Trees) *Suppose T is a type.*

- *A leaf is a binary tree over T.*
- *If $t \in T$ and B_1, B_2 are binary trees over T, then node(t, B_1, B_2) is a binary tree over T.*

BinTree(T) is the type of all binary trees over T.

According to this definition, leafs do not carry information (i.e. elements from T). All information is stored in the nodes, and in the structure of the tree itself.

The Nuprl abstraction defining binary trees is shown in Figure 1. Due to the use of the disjoint product type $+$, a binary tree in Nuprl now is equal to either *inl* $\cdot$ or *inr* $< t, B_1, B_2 >$, where $t \in T$ and B_1, B_2 are binary trees. We define `leaf` as an abbreviation for *inl* $\cdot$, and `node(t,B_1,B_2)` as an abbreviation for *inr* $< t, B_1, B_2 >$, as shown in Figure 2. The fact that `leaf` and `node(t,B_1,B_2)` are binary trees is captured by the two well-formedness theorems shown in Figure 3. The theorems are proved in two steps each.

* This work was supported by NSF grant CCR-9985239 and a DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research under grant N00014-01-1-0765.

```
* ABS binary_tree
BinTree(T) == rec(t.Unit + T × t × t)
```

Fig. 1. Abstraction `binary_tree`

```
* ABS leaf
leaf == inl ·

* ABS node
node(t,b1,b2) == inr <t, b1, b2>
```

Fig. 2. Abstractions `leaf` and `node`

```
* THM leaf_wf
∀T:𝕌. leaf ∈ BinTree(T)

* THM node_wf
∀T:𝕌. ∀t:T. ∀B1,B2:BinTree(T). node(t,B1,B2) ∈ BinTree(T)
```

Fig. 3. Well-formedness theorems for `leaf` and `node`

3 The *reduce* Operator

Suppose T and R are types, $c \in R$ and $g : T \times R \times R \to R$. We want to define a function $f : \mathit{BinTree}(T) \to R$ by the following recursion over binary trees:

$$f(\mathit{leaf}) = c$$
$$f(\mathit{node}(t, B_1, B_2)) = g(t, f(B_1), f(B_2))$$

The *reduce* operator is defined such that $f = \mathit{reduce}(c; g)$.

Definition 2 (reduce) *Suppose T and R are types, $c \in R$ and $g : T \times R \times R \to R$. Define $\mathit{reduce}(c; g) : \mathit{BinTree}(T) \to R$ recursively by*

$$\mathit{reduce}(c; g)(B) = \begin{cases} c & \text{if } B = \mathit{leaf} \\ g(t, \mathit{reduce}(c; g)(B_1), \mathit{reduce}(c; g)(B_2)) & \text{if } B = \mathit{node}(t, B_1, B_2) \end{cases}$$

for all $B \in \mathit{BinTree}(T)$.

The corresponding abstraction `treereduce` is shown in Figure 4. We use a curried function $g : T \to R \to R \to R$ in the `treereduce` abstraction instead of a function with domain $T \times R \times R$ and codomain R. Avoiding the cartesian product (and consequently, tuples as function arguments) simplifies the Nuprl notation.

Since *reduce* is defined recursively, we have to verify that this recursion always terminates to make sure that $\mathit{reduce}(c; g)$ is well-defined, i.e. that $\mathit{reduce}(c; g)(B)$ is in R for all binary trees B.

Lemma 1. *Suppose T and R are types, $c \in R$ and $g : T \times R \times R \to R$. Then*

$$\mathit{reduce}(c; g)(B) \in R$$

for all $B \in \mathit{BinTree}(T)$.

```
* ABS treereduce
reduce(c;g)(B) ==
    (letrec recfun(B) =
        case B of
          inl(x) => c
        | inr(y) => let t,B1,B2 = y in g t (recfun B1) (recfun B2))
    B
```

Fig. 4. Abstraction `treereduce`

Proof. Let $B \in \mathit{BinTree}(T)$. We use structural induction on B.

Base case ($B = \mathit{leaf}$): $\mathit{reduce}(c; g)(B) = c \in R$.

Inductive step ($B = \mathit{node}(t, B_1, B_2)$): By the induction hypothesis, $\mathit{reduce}(c; g)(B_1) \in R$ and $\mathit{reduce}(c; g)(B_2) \in R$. Therefore

$$\mathit{reduce}(c; g)(B) = g(t, \mathit{reduce}(c; g)(B_1), \mathit{reduce}(c; g)(B_2)) \in R.$$

The proof of the formal theorem `treereduce_wf`, which is shown in Figure 5, proceeds along the same lines. The RecElimination tactic is used for structural induction on B. The base case is then proved by the Auto tactic after we unfold the definition of `treereduce`. The induction hypothesis is used to prove the inductive step. Altogether the proof is about nine steps long.

```
* THM treereduce_wf
∀T,R:𝕌.∀c:R.∀g:T → R → R → R. ∀B:BinTree(T). reduce(c;g)(B) ∈ R
```

Fig. 5. Theorem `treereduce_wf`

Example 1. The *height* of a binary tree (over an arbitrary type T) can be defined recursively. The height of a leaf is 0, and the height of a node is one more than the maximum of the heights of the node's left and right subtree:

$$\begin{aligned} \mathit{height}(\mathit{leaf}) &= 0, \\ \mathit{height}(\mathit{node}(t, B_1, B_2)) &= 1 + \max(\mathit{height}(B_1), \mathit{height}(B_2)). \end{aligned}$$

See Figure 6 for a formal definition.

```
* ABS treeheight
|B|==
   (letrec recfun(B) =
       case B of
         inl(x) => 0
       | inr(y) => let t,B1,B2 = y in 1 + imax(recfun B1;recfun B2) )
   B
```

Fig. 6. Abstraction `treeheight`

Clearly $height(B) \in \mathbb{N}$ for all binary trees B; this fact is proved by the theorem `treeheight_wf` shown in Figure 7. Again the RecElimination tactic is used for structural induction on B in the proof of this theorem. The formal proof is about 27 steps long, mainly because we have to overcome a few technical difficulties caused by the use of $\mathbb{N}$ and $\mathbb{Z}$.

```
* THM treeheight_wf
∀T:𝕌. ∀B:BinTree(T). |B| ∈ ℕ
```

Fig. 7. Theorem `treeheight_wf`

Alternatively, *height* can be defined in terms of *reduce*. Define $g : T \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by $g(t,m,n) = 1 + \max(m,n)$. Then $height(B) = reduce(0; g)(B)$ for all binary trees B, as shown in Figure 8. The proof of this theorem is about ten steps long and uses both the RecElimination tactic and the `treeheight_wf` lemma, as well as a few other lemmata.

```
* THM treereduce_example
∀T:𝕌. ∀B:BinTree(T). |B| = reduce(0;λt,m,n.1 + imax(m;n))(B)
```

Fig. 8. Theorem `treereduce_example`

4 The *unfold* Operator

The *reduce* operator extracts some information from a binary tree. It provides a general pattern to define catamorphisms on binary trees. Now suppose S is a type, and we want to define an operator *unfold* that *constructs* a binary tree from some input $x \in S$ as follows. First, a given predicate p is applied to x. If $p(x)$ is true, we apply a function f to x that computes a node value a and two new input values y and z. *unfold* is then recursively applied to y and z to compute the left and right subtree of the node. If $p(x)$ is false, *unfold* simply returns *leaf*.

However, there is a problem with this 'definition'. If y and z are allowed to be arbitrary input values, this recursion is not guaranteed to terminate: Assume $p(x)$ is true for every input x, and consider the function $f : S \to S \times S \times S$, defined by $f(x) = (x, x, x)$. Then

$$\begin{aligned} unfold(p;f)(x) &= node(x, unfold(p;f)(x), unfold(p;f)(x)) \\ &= node(x, \\ &\qquad node(x, unfold(p;f)(x), unfold(p;f)(x)), \\ &\qquad node(x, unfold(p;f)(x), unfold(p;f)(x))) \\ &= \ldots \end{aligned}$$

To guarantee that the recursion terminates, we require y and z to be 'smaller' than x, where the 'size' of an input is just a natural number.[1]

[1] As pointed out by N. Shankar [7], any well-founded ordering could be used here instead of the less-than relation on natural numbers.

Definition 3 (Smaller) *Suppose S is a type, $size : S \to \mathbb{N}$, and $x \in S$. Then we define*

$$Smaller(S, size, x) = \{y \in S \mid size(y) < size(x)\}$$

to be the type of all elements in S with a size less than the size of x.

The formal definition of `Smaller` is shown in Figure 9. The well-formedness theorem `smaller_wf` proves that `Smaller(S,size,x)` is a type if S is a type, $size : S \to \mathbb{N}$, and $x \in S$. It is proved in a single step by the AUTO tactic.

```
* ABS smaller
Smaller(S,size,x) == {y:S| size y < size x}
```

Fig. 9. Abstraction `Smaller`

Now we are ready to define the type of functions that we allow as a parameter to *unfold*. Note that to compute $unfold(p; f)(x)$, we only need to evaluate $f(x)$ when $p(x)$ is true. Therefore the domain of f does not need to be S, but it can be restricted to the subtype $\{x \in S \mid p(x) = \text{true}\}$.

Definition 4 (WellFnd) *Suppose S and T are types, $p : S \to \mathbb{B}$, and $size : S \to \mathbb{N}$. Then we define*

$$\begin{aligned}
&WellFnd(S, p, size, T) = \\
&\quad \{f : \{x \in S \mid p(x) = true\} \to T \times S \times S \mid \\
&\qquad \forall x \in \{x \in S \mid p(x) = true\} : \\
&\qquad\quad f(x) \in T \times Smaller(S, size, x) \times Smaller(S, size, x)\}.
\end{aligned}$$

In NUPRL, the dependent function type can be used to define *WellFnd* more elegantly: The codomain does not have to be a single type, but it can depend on the function argument x. Thus given x, we can require $f(x)$ to be in $T \times Smaller(S, size, x) \times Smaller(S, size, x)$. Figure 10 shows the corresponding abstraction `treewellfnd`.

```
* ABS treewellfnd
WellFnd(S,p,size,T) ==
x:{x:S| p[x] = tt} → (T × Smaller(S,size,x) × Smaller(S,size,x))
```

Fig. 10. Abstraction `treewellfnd`

The well-formedness theorem for `treewellfnd` simply states that this is a type if S and T are types, $p : S \to \mathbb{B}$, and $size : S \to \mathbb{N}$. It is proved in a single step by NUPRL's AUTO tactic. Using the type *WellFnd* of *'well-founded'* functions, we can now precisely define *unfold*.

Definition 5 (unfold) *Suppose S and T are types, $p : S \to \mathbb{B}$, $size : S \to \mathbb{N}$, and $f \in WellFnd(S, p, size, T)$. Define $unfold(p; f) : S \to BinTree(T)$ recursively by*

$$unfold(p; f)(x) = \begin{cases} node(a, unfold(p; f)(y), unfold(p; f)(z)) & \text{if } p(x) \text{ is true} \\ leaf & \text{if } p(x) \text{ is false} \end{cases}$$

for all $x \in S$, where $f(x) = (a, y, z)$.

```
* ABS treeunfold
unfold(p;f)(x) ==
  (letrec recfun(x) =
      if p[x] then
        let a,y,z = (f x) in node(a; recfun y; recfun z)
      else
        leaf
      fi )
  x
```

Fig. 11. Abstraction `treeunfold`

See Figure 11 for the definition of `treeunfold` in Nuprl.

The restrictions imposed on f allow us to prove that *unfold* is well-defined, i.e. that the recursion always terminates.

Lemma 2. *Suppose S and T are types, $p : S \to \mathbb{B}$, $size : S \to \mathbb{N}$, and $f \in WellFnd(S, p, size, T)$. Then*

$$unfold(p; f)(x) \in BinTree(T)$$

for all $x \in S$.

Proof. Let $x \in S$. We show $unfold(p; f)(x) \in BinTree(T)$ by complete induction on $size(x)$. Assume $unfold(p; f)(y) \in BinTree(T)$ for all $y \in S$ with $size(y) < size(x)$.

Case 1: Assume $p(x)$ is false. Then $unfold(p; f)(x) = leaf \in BinTree(T)$.

Case 2: Assume $p(x)$ is true. Let $f(x) = (a, y, z)$. Then $y, z \in Smaller(S, size, x)$ since $f \in WellFnd(S, p, size, T)$. Hence $size(y) < size(x)$ and $size(z) < size(x)$. Thus $unfold(p; f)(y) \in BinTree(T)$ and $unfold(p; f)(z) \in BinTree(T)$ by the induction hypothesis. Therefore

$$unfold(p; f)(x) = node(a, unfold(p; f)(y), unfold(p; f)(z)) \in BinTree(T).$$

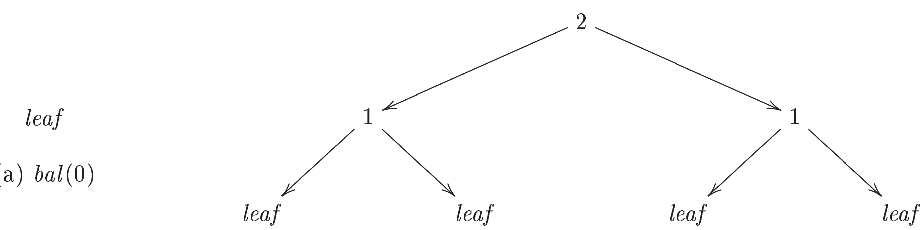
In Nuprl we state this lemma as a well-formedness theorem for `treeunfold`. This well-formedness theorem is shown in Figure 12. The formal proof uses Nuprl's InvImageInd tactic in combination with the CompNatInd tactic for complete induction on the size of x. The IfThenElse tactic is then used for the case split on $p(x)$. The proof is about 15 steps long.

```
* THM treeunfold_wf
∀S:𝕌.∀p:S → 𝔹.∀size:S → ℕ.∀T:𝕌.∀f:WellFnd(S,p,size,T).∀x:S.
   unfold(p;f)(x) ∈ BinTree(T)
```

Fig. 12. Theorem `treeunfold_wf`

The *unfold* operator, just like *reduce*, can be used to specify a number of algorithms. We give a simple example below, and a more elaborate example in the following section.

Example 2. We say a binary tree B is *balanced* if and only if every leaf in B has the same height. Consider a function $bal : \mathbb{N} \to BinTree(\mathbb{N})$ that, given a natural number n, creates a balanced binary tree of height n in which every node is labelled with its height (i.e. the root node is labelled with n, the two nodes directly beneath it are labelled with $n - 1$, and so on). See Figure 13 for two examples.



(a) $bal(0)$

(b) $bal(2)$

Fig. 13. Example: *bal*

More precisely, let $bal : \mathbb{N} \to BinTree(\mathbb{N})$ be defined inductively by

$$
\begin{aligned}
bal(0) &= leaf, \\
bal(n+1) &= node(n+1, bal(n), bal(n)).
\end{aligned}
$$

The NUPRL abstraction defining *bal* is shown in Figure 14. The well-formedness theorem `create_balanced_wf` proves that `create_balanced(n)` is in $BinTree(\mathbb{N})$ for every $n \in \mathbb{N}$. We use the NATIND tactic in the proof of `create_balanced_wf` for mathematical induction on n. The proof is about six steps long.

```
* ABS create_balanced
create_balanced(n) ==
  (letrec recfun(n) =
      if(n =z 0) then
        leaf
      else
        node(n; recfun (n - 1); recfun (n - 1))
      fi)
  n
```

Fig. 14. Abstraction `create_balanced`

Now define $p : \mathbb{N} \to \mathbb{B}$ by $p(n) \iff (n \neq 0)$, and define $g : \mathbb{N} \setminus \{0\} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ by $g(n) = (n, n-1, n-1)$. Then $bal(n) = unfold(p; g)(n)$ for all $n \in \mathbb{N}$, as proved by the theorem `treeunfold_example` shown in Figure 15. The proof uses NUPRL's NATIND tactic for mathematical induction on n. It is about 92 steps long, mainly because several well-formedness goals need to be verified.

```
* THM treeunfold_example
∀n:ℕ. create_balanced(n) = unfold((λn.¬b(n =z 0)); (λn.<n, n - 1, n - 1>); n)
```

Fig. 15. Theorem `treeunfold_example`

5 The *fun* Operator

The composition of *unfold* and *reduce* can be used to specify a large number of algorithms, e.g. the QUICKSORT algorithm (see Section 9 for details). However, *unfold* first constructs a binary tree, and *reduce* then consumes the tree. Bird's fusion transformation allows us to replace $\mathit{reduce} \cdot \mathit{unfold}$ with a single operator *fun* (defined below) that does not construct an intermediate tree. This is an instance of *deforestation* [3,8,5], a program optimization technique that fuses adjacent phases to eliminate the intermediate data structures.

Definition 6 (fun) *Suppose* S, T, R *are types,* $p : S \to \mathbb{B}$*,* $\mathit{size} : S \to \mathbb{N}$*, and* $f \in \mathit{WellFnd}(S, p, \mathit{size}, T)$*. Furthermore, suppose* $c \in R$ *and* $g : T \times R \times R \to R$*. Define* $\mathit{fun}(p; f; c; g) : S \to R$ *by*

$$\mathit{fun}(p; f; c; g)(x) = \begin{cases} g(a, \mathit{fun}(p; f; c; g)(y), \mathit{fun}(p; f; c; g)(z)) & \text{if } p(x) \text{ is true} \\ c & \text{if } p(x) \text{ is false} \end{cases}$$

for all $x \in S$*, where* $f(x) = (a, y, z)$*.*

Figure 16 shows the corresponding NUPRL abstraction `treefun`. Again we avoid tuples as function arguments by using a curried function g.

```
* ABS treefun
fun(p;f;c;g)(x) ==
  (letrec recfun(x) =
      if p[x] then
        let a,y,z = (f x) in g a (recfun y) (recfun z)
      else c
      fi )
  x
```

Fig. 16. Abstraction `treefun`

The operator *fun*, like *reduce* and *unfold* before, is defined recursively. Therefore we need to verify that it is well-defined, i.e. that the recursion terminates for every input $x \in S$.

Lemma 3. *Suppose* S, T, R *are types,* $p : S \to \mathbb{B}$*,* $\mathit{size} : S \to \mathbb{N}$*, and* $f \in \mathit{WellFnd}(S, p, \mathit{size}, T)$*. Furthermore, suppose* $c \in R$ *and* $g : T \times R \times R \to R$*. Then*

$$\mathit{fun}(p; f; c; g)(x) \in R$$

for all $x \in S$*.*

Proof. Let $x \in S$. We show $\mathit{fun}(p; f; c; g)(x) \in R$ by complete induction on $\mathit{size}(x)$. Assume $\mathit{fun}(p; f; c; g)(y) \in R$ for all $y \in S$ with $\mathit{size}(y) < \mathit{size}(x)$.

Case 1: Assume $p(x)$ is false. Then $\mathit{fun}(p; f; c; g)(x) = c \in R$.

Case 2: Assume $p(x)$ is true. Let $f(x) = (a, y, z)$. Then $y, z \in \mathit{Smaller}(S, \mathit{size}, x)$ since $f \in \mathit{WellFnd}(S, p, \mathit{size}, T)$. Hence $\mathit{size}(y) < \mathit{size}(x)$ and $\mathit{size}(z) < \mathit{size}(x)$. Thus $\mathit{fun}(p; f; c; g)(y) \in R$ and $\mathit{fun}(p; f; c; g)(z) \in R$ by the induction hypothesis. Therefore

$$\mathit{fun}(p; f; c; g)(x) = g(a, \mathit{fun}(p; f; c; g)(y), \mathit{fun}(p; f; c; g)(z)) \in R.$$

The formal well-formedness theorem is shown in Figure 17. Its proof is about eleven steps long and uses the INVIMAGEIND tactic in combination with COMPNATIND for complete induction on the size of x.

```
* THM treefun_wf
∀S:𝕌.∀p:S → 𝔹.∀size:S → ℕ.∀T:𝕌.∀f:WellFnd(S,p,size,T).
   ∀R:𝕌.∀c:R.∀g:T → R → R → R.∀x:S.
      fun(p;f;c;g)(x) ∈ R
```

Fig. 17. Theorem `treefun_wf`

6 Bird's Fusion Theorem for Binary Trees

As mentioned before, we want to replace *reduce* $\cdot$ *unfold* with *fun* to eliminate the intermediate tree. In this section we prove that *reduce* $\cdot$ *unfold* and *fun* are equivalent, in the sense that they compute the same function.

Theorem 7 (Bird's Fusion Theorem for Binary Trees). *Suppose S, T, R are types, $p : S \to \mathbb{B}$, size : $S \to \mathbb{N}$, and $f \in \mathit{WellFnd}(S, p, \mathit{size}, T)$. Furthermore, suppose $c \in R$ and $g : T \times R \times R \to R$. Then*

$$(\mathit{reduce}(c; g) \cdot \mathit{unfold}(p; f))(x) = \mathit{fun}(p; f; c; g)(x)$$

for all $x \in S$.

Proof. Let $x \in S$. We show $(\mathit{reduce}(c; g) \cdot \mathit{unfold}(p; f))(x) = \mathit{fun}(p; f; c; g)(x)$ by complete induction on $\mathit{size}(x)$. Assume $(\mathit{reduce}(c; g) \cdot \mathit{unfold}(p; f))(y) = \mathit{fun}(p; f; c; g)(y)$ for all $y \in S$ with $\mathit{size}(y) < \mathit{size}(x)$.

Case 1: Assume $p(x)$ is false. Then

$$\begin{aligned}(\mathit{reduce}(c; g) \cdot \mathit{unfold}(p; f))(x) &= \mathit{reduce}(c; g)(\mathit{unfold}(p; f)(x))\\ &= \mathit{reduce}(c; g)(\mathit{leaf})\\ &= c\\ &= \mathit{fun}(p; f; c; g)(x).\end{aligned}$$

Case 2: Assume $p(x)$ is true. Let $f(x) = (a, y, z)$. Then $y, z \in \mathit{Smaller}(S, \mathit{size}, x)$ since $f \in \mathit{WellFnd}(S, p, \mathit{size}, T)$. Hence $\mathit{size}(y) < \mathit{size}(x)$ and $\mathit{size}(z) < \mathit{size}(x)$. Thus $(\mathit{reduce}(c; g) \cdot \mathit{unfold}(p; f))(y) = \mathit{fun}(p; f; c; g)(y)$ and $(\mathit{reduce}(c; g) \cdot \mathit{unfold}(p; f))(z) = \mathit{fun}(p; f; c; g)(z)$ by the induction hypothesis. Therefore

$$\begin{aligned}&(\mathit{reduce}(c; g) \cdot \mathit{unfold}(p; f))(x)\\ &\quad = \mathit{reduce}(c; g)(\mathit{unfold}(p; f)(x))\\ &\quad = \mathit{reduce}(c; g)(\mathit{node}(a, \mathit{unfold}(p; f)(y), \mathit{unfold}(p; f)(z)))\\ &\quad = g(a, \mathit{reduce}(c; g)(\mathit{unfold}(p; f)(y)), \mathit{reduce}(c; g)(\mathit{unfold}(p; f)(z)))\\ &\quad = g(a, (\mathit{reduce}(c; g) \cdot \mathit{unfold}(p; f))(y), (\mathit{reduce}(c; g) \cdot \mathit{unfold}(p; f))(z))\\ &\quad = g(a, \mathit{fun}(p; f; c; g)(y), \mathit{fun}(p; f; c; g)(z))\\ &\quad = \mathit{fun}(p; f; c; g)(x)\end{aligned}$$

as required.

Figure 18 shows the formal `fusion` theorem. The proof uses the usual combination of the tactics InvImageInd and CompNatInd for complete induction on the size of x; it is about 27 steps long.

In the following sections we apply the fusion transformation to the Quicksort algorithm.

```
* THM fusion
∀S:𝕌. ∀p:S → 𝔹. ∀size:S → ℕ. ∀T:𝕌. ∀f:WellFnd(S,p,size,T).
   ∀Range:𝕌. ∀c:Range. ∀g:T → Range → Range → Range. ∀x:S.
      reduce(c;g)(unfold(p;f)(x)) = fun(p;f;c;g)(x)
```

Fig. 18. Theorem `fusion`

7 Quicksort

The QUICKSORT algorithm was first published by C.A.R. Hoare [6] in 1961. It is "one of the fastest, the best known, the most generalized, ... and the most widely used algorithms for sorting an array of numbers" [4]. Both R. Bird [1] and N. Shankar [7] chose it as an example to apply the fusion transformation to.

Despite its speed, QUICKSORT is a relatively simple algorithm. It can be described as follows.

1. If the list is empty, there is nothing to do.
2. Otherwise pick an element from the list to be the 'partition element'.
3. Divide the other elements into those less than or equal to the partition element, and those greater than the partition element.
4. Arrange the elements in the list such that the order is the elements below the partition element, the partition element itself, and the elements above the partition element.
5. Recursively invoke QUICKSORT on the smaller elements.
6. Recursively invoke QUICKSORT on the larger elements.

As we can see from this description, QUICKSORT can be used for any type on which an order relation $\leq$ is defined.[2]

8 Quicksort in Nuprl

Figure 19 shows an implementation of the QUICKSORT algorithm in NUPRL. We define `quicksort` as a recursive function that takes a relation $\leq$ and a list L as arguments and returns a list (NUPRL's built-in data type `list` is used here). If L is the empty list, denoted as [], then the empty list is returned. Otherwise the head of L is picked as the partition element. Then `quicksort` is invoked recursively on a list of all elements in the tail of L that are smaller than or equal to ('`below`') the head of L, and on a list of all elements in the tail of L that are larger than ('`above`') the head of L. Both lists are generated by the `filter` function: `filter(p;L)` returns a list with those elements in L that satisfy the predicate p. Finally `append` (`@`) and `cons` (`::`) are used to concatenate the two lists and the partition element in the proper order.

The `quicksort` function is defined recursively. We prove that it is well-defined by complete induction on the length of the input list L.

Lemma 4. *Suppose T is a type and $\leq : T \times T \to \mathbb{B}$. Then*

$$quicksort(\leq, L) \in List(T)$$

for all $L \in List(T)$.

We first prove another lemma, namely that the list returned by $filter(p; L)$ is at most as long as L.

[2] Note that even when $\leq$ is not an order relation, we can still formally apply QUICKSORT. In fact, we will prove that QUICKSORT returns a permutation of its input when $\leq$ is an arbitrary relation on the type of the list elements.

```
* ABS quicksort
quicksort(≤,L) ==
  (letrec recfun(L) =
    case L of
      [] => []
    | a::y => (recfun filter(λb.b below(≤) a; y))
                @ (a::(recfun filter(λb.b above(≤) a; y)))
    esac)
  L
```

Fig. 19. Abstraction `quicksort`

Lemma 5. *Suppose T is a type, and $f : T \to \mathbb{B}$. Then*

$$|\mathit{filter}(f, L)| \leq |L|$$

for all $L \in \mathit{List}(T)$.

The `filter` abstraction is part of the `LIST_3` library, as is the abstraction defining `list_length`. Here we define *filter* as follows.

Definition 8 (filter) *Suppose T is a type, $f : T \to \mathbb{B}$, and $L \in \mathit{List}(T)$. Define $\mathit{filter}(f; L) \in \mathit{List}(T)$ recursively by*

$$\mathit{filter}(f; L) = \begin{cases} [] & \text{if } L = [] \\ \mathit{filter}(f; t) & \text{if } L = h :: t \text{ and } f(h) \text{ is false} \\ h :: \mathit{filter}(f; t) & \text{if } L = h :: t \text{ and } f(h) \text{ is true} \end{cases}.$$

With this definition we can easily prove Lemma 5.

Proof. The proof is by structural induction on L.

Base case ($L = []$): $|\mathit{filter}(f; L)| = |[]| = |L|$.

Inductive step ($L = h :: t$): By the induction hypothesis, $|\mathit{filter}(f; t)| \leq |t|$. If $f(h) = \text{true}$,

$$|\mathit{filter}(f; L)| = |h :: \mathit{filter}(f; t)| = 1 + |\mathit{filter}(f; t)| \leq 1 + |t| = |L|.$$

If $f(h) = \text{false}$,

$$|\mathit{filter}(f; L)| = |\mathit{filter}(f; t)| \leq |t| = |L| - 1 < |L|.$$

Figure 20 shows the corresponding Nuprl theorem `list_length_filter`. The proof of the formal theorem uses the ListInd tactic for structural induction on L, and the IfThenElseCases tactic for the case split on $f(h)$. The proof is about six steps long; most of the work is done by Nuprl's Auto tactic.

```
* THM list_length_filter
∀T:𝕌. ∀f:T → 𝔹. ∀L:T List. |·| filter(f;L) ≤ |·| L
```

Fig. 20. Theorem `list_length_filter`

Given a type T and a relation $\leq\, : T \times T \to \mathbb{B}$, we define b *below*$(\leq)$ a as $b \leq a$, and b *above*$(\leq)$ a as $\neg(b$ *below*$(\leq)$ $a)$ for a, $b \in T$. The corresponding Nuprl abstractions `below` and `above` are shown in Figure 21.

```
* ABS below
b below(≤) a == b ≤ a

* ABS above
b above(≤) a == ¬_b b below(≤) a
```

Fig. 21. Abstractions `below` and `above`

The well-formedness theorems `below_wf` and `above_wf` prove that $b\ below(\leq)\ a$ and $b\ above(\leq)\ a$ are in $\mathbb{B}$ if T is a type, $\leq : T \times T \to \mathbb{B}$, and a, $b \in T$. They are proved in a single step each. Now we are ready to prove Lemma 4.

Proof. By complete induction on the length of L. Assume $quicksort(\leq, M) \in List(T)$ for all $M \in List(T)$ with $|M| < |L|$.

Case 1: Assume $L = []$. Then $quicksort(\leq, L) = [] \in List(T)$.

Case 2: Assume $L = h :: t$, where $h \in T$ and $t \in List(T)$. By Lemma 5, $|filter(b\ below(\leq)\ h; t)| \leq |t| < |L|$ and $|filter(b\ above(\leq)\ h; t)| \leq |t| < |L|$. Thus

$$quicksort(\leq, filter(b\ below(\leq)\ h; t)) \in List(T)$$

and

$$quicksort(\leq, filter(b\ above(\leq)\ h; t)) \in List(T)$$

by the induction hypothesis. Therefore

$$\begin{aligned} & quicksort(\leq, L) \\ & = quicksort(\leq, filter(b\ below(\leq)\ h; t)) \\ & \qquad @\ (h :: quicksort(\leq, filter(b\ above(\leq)\ h; t))) \\ & \in List(T). \end{aligned}$$

The NUPRL theorem `quicksort_wf` is shown in Figure 22. Note the use of a curried function $\leq : T \to T \to \mathbb{B}$ to avoid tuples as function arguments. The formal proof uses the LISTLENIND tactic for complete induction on the length of the list L. Then CASES is used to do a case split on $L = []$ and $L = h :: t$. The case $L = []$ is proved by an invocation of the LISTIND tactic, because even though we know that L is equal to $[]$, we cannot substitute $[]$ for L in the proof goal $quicksort(\leq, L) \in List(T)$ without creating unprovable well-formedness goals. For the same reason, we cannot simply substitute $h :: t$ for L in the other case. We circumvent this problem by eliminating L from all hypotheses first (by substituting $h :: t$ for L, or by moving them to the conclusion), and by decomposing the declaration of L as a list then. With 26 steps altogether, the proof is relatively short, but surprisingly tricky.

```
* THM quicksort_wf
∀T:𝕌. ∀≤:T → T → 𝔹. ∀L:T List. quicksort(≤,L) ∈ T List
```

Fig. 22. Theorem `quicksort_wf`

9 Quicksort by Fusion

If we compare our implementation of QUICKSORT (Figure 19) to the `treefun` operator (Figure 16) defined in the previous section, it is almost obvious that QUICKSORT can be written as `treefun`, and hence—by the fusion theorem—that QUICKSORT is equal to the composition of an anamorphism and a catamorphism. In this section we make a few necessary definitions before we finally prove this equality.

Using a binary tree, we can split QUICKSORT into two phases. The first phase constructs an ordered binary tree that contains the same elements as the input list L as follows: The partition element becomes the tree's root value. The left subtree and the right subtree are recursively constructed from a list of those elements in the tail of L that are below the partition element, and from a list of those elements in L that are above the partition element. The empty list [] simply becomes a leaf.

The second phase flattens the ordered binary tree into an ordered list by an in-order search: First the left subtree is flattened, then the root value is inserted at the end of the list, then the right subtree is flattened.

Flattening a binary tree is a catamorphism that can easily be defined in terms of *reduce*.

Definition 9 (flatten) *Suppose T is a type. Let $g : T \times \mathit{List}(T) \times \mathit{List}(T) \to \mathit{List}(T)$ be defined by $g(a, x, y) = x@(a :: y)$. Define $\mathit{flatten} : \mathit{BinTree}(T) \to \mathit{List}(T)$ by*

$$\mathit{flatten}(B) = \mathit{reduce}([]; g)(B).$$

The formal definition of `flatten` is shown in Figure 23. The well-formedness theorem `flatten_wf` proves that $\mathit{flatten}(B)$ is a list over T for every type T and every $B \in \mathit{BinTree}(T)$. It is proved in two steps by instantiating the `treereduce_wf` lemma.

```
* ABS flatten
flatten(B) == reduce([];λa,x,y.x @ (a::y))(B)

* THM flatten_wf
∀T:𝕌. ∀B:BinTree(T). flatten(B) ∈ T List
```

Fig. 23. Abstraction `flatten` and Theorem `flatten_wf`

Defining the first phase of QUICKSORT in terms of *unfold* requires a little more effort. Firstly we define a simple predicate $\mathit{is\_cons} : \mathit{List}(T) \to \mathbb{B}$ such that $\mathit{is\_cons}(L)$ is true if and only if $L = h :: t$ for some $h \in T$, $t \in \mathit{List}(T)$. The abstraction `is_cons` is shown in Figure 24.

```
* ABS is_cons
is_cons == λL.case L of [] => ff | h::t => tt esac
```

Fig. 24. Abstraction `is_cons`

The well-formedness theorem `is_cons_wf` states that $\mathit{is\_cons} : \mathit{List}(T) \to \mathbb{B}$ for every type T. It is proved in a single step by the AUTO tactic. We also prove two useful lemmata, namely that $\mathit{is\_cons}([])$ is false and that $\mathit{is\_cons}(h :: t)$ is true (see Figure 25). The lemmata are proved in a single step each by unfolding the definition of `is_cons` and applying the AUTO tactic afterwards.

We then define a function $\mathit{unjoin}(\leq) : \{L \in \mathit{List}(T) \mid \mathit{is\_cons}(L)\} \to T \times \mathit{List}(T) \times \mathit{List}(T)\}$ that maps a non-empty list L to the triple that has $\mathit{hd}(L)$ as its first component, the list of all elements in $\mathit{tl}(L)$ that

```
* THM is_cons_of_nil
is_cons [] = ff

* THM is_cons_of_cons
∀T:𝕌. ∀u:T. ∀v:T List. is_cons (u::v) = tt
```

Fig. 25. Theorems `is_cons_of_nil` and `is_cons_of_cons`

are below $hd(L)$ as its second element, and finally the list of all elements in $tl(L)$ that are above $hd(L)$ as its third element.

Definition 10 (unjoin) *Suppose T is a type and $\leq : T \times T \to \mathbb{B}$. Define $unjoin(\leq) : \{L \in List(T) \mid is\_cons(L)\} \to T \times List(T) \times List(T)$ by*

$$\begin{aligned} &unjoin(\leq)(L) = \\ &\quad (hd(L), filter(\cdot\ below(\leq)\ hd(L); tl(L)), filter(\cdot\ above(\leq)\ hd(L); tl(L))) \end{aligned}$$

for all $L \in List(T)$ with $is\_cons(L) = true$.

The NUPRL abstraction `unjoin` is shown in Figure 26. We want to use *unjoin* as an argument to the *unfold* operator defined in Section 1, so we have to verify that *unjoin* is a 'well-founded' function.

```
* ABS unjoin
unjoin(≤) ==
  λx.<hd(x),
       filter((λb.b below(≤) hd(x));tl(x)),
       filter((λb.b above(≤) hd(x));tl(x))>
```

Fig. 26. Abstraction `unjoin`

Lemma 6. *Suppose T is a type and $\leq : T \times T \to \mathbb{B}$. Then*

$$unjoin(\leq) \in WellFnd(List(T), is\_cons, |\cdot|, T).$$

Proof. Clearly $unjoin(\leq) : \{L \in List(T) \mid is\_cons(L)\} \to T \times List(T) \times List(T)$. We have to verify

$$filter(\cdot\ below(\leq)\ hd(L); tl(L)) \in Smaller(List(T), |\cdot|, L)$$

and

$$filter(\cdot\ above(\leq)\ hd(L); tl(L)) \in Smaller(List(T), |\cdot|, L)$$

for all L in $List(T)$ with $is\_cons(L) =$ true.

Both statements follow from Lemma 5 in combination with $|tl(L)| = |L| - 1 < |L|$.

We prove this lemma as a well-formedness theorem `unjoin_wf` in NUPRL (see Figure 27). The formal proof is about 24 steps long. It uses a number of lemmata, including `list_length_filter` and `length_tl`. The latter proves $|tl(L)| = |L| - 1$. It can be found in the `LIST_1` library. The final proof step for each of the two statements invokes the SUPINF tactic which handles integer inequalities in NUPRL.

We can now define a function $mktree(\leq) : List(T) \to BinTree(T)$ that implements the first phase of QUICKSORT, that is, the generation of an ordered binary tree from a list.

```
* THM unjoin_wf
∀T:𝕌. ∀≤:T → T → 𝔹. unjoin(≤) ∈ WellFnd(T List,is_cons,|·|,T)
```

Fig. 27. Theorem `unjoin_wf`

Definition 11 (mktree) *Suppose T is a type, and $\leq : T \times T \to \mathbb{B}$. Define $mktree(\leq) : List(T) \to BinTree(T)$ by*

$$mktree(\leq)(L) = unfold(is\_cons; unjoin(\leq))(L)$$

for all $L \in List(T)$.

The `mktree` abstraction and the associated well-formedness theorem `mktree_wf` are shown in Figure 28. The well-formedness theorem is proved in a single step by the AUTO tactic.

```
* ABS mktree
mktree(≤)(x) == unfold(is_cons;unjoin(≤))(x)

* THM mktree_wf
∀T:𝕌. ∀≤:T → T → 𝔹. ∀L:T List. mktree(≤)(L) ∈ BinTree(T)
```

Fig. 28. Abstraction `mktree` and Theorem `mktree_wf`

Like for `is_cons` before, we prove two simple, yet useful lemmata about `mktree` that can later be used when we do structural induction on a list L. The first lemma proves $mktree(\leq)([]) = leaf$, and the second lemma proves $mktree(\leq)(u :: v) = node(u, mktree(\leq)(filter(\cdot\ below(\leq)\ u; v)), mktree(\leq)(filter(\cdot\ above(\leq)\ u; v))$. The lemmata are shown in Figure 29. The proof of `mktree_of_nil` is about seven steps long, and proving `mktree_of_cons` requires about nine steps—mainly just unfolding definitions.

```
* THM mktree_of_nil
∀T:𝕌. ∀≤:T → T → 𝔹. mktree(≤)([]) = leaf

* THM mktree_of_cons
∀T:𝕌. ∀≤:T → T → 𝔹. ∀u:T. ∀v:T List.
   mktree(≤)(u::v) = node(u,
                          mktree(≤)(filter((λb.b below(≤) u);v)),
                          mktree(≤)(filter((λb.b above(≤) u);v)))
```

Fig. 29. Theorems `mktree_of_nil` and `mktree_of_cons`

We have a second way of stating the QUICKSORT algorithm now: *quicksort* is equal to the composition of *mktree* and *flatten*.

Theorem 12. *Suppose T is a type and $\leq : T \times T \to \mathbb{B}$. Then*

$$quicksort(\leq, L) = flatten(mktree(\leq)(L))$$

for all $L \in List(T)$.

The theorem `quicksort_by_fusion` shown in Figure 30 formalizes this result in NUPRL. To prove it, we first replace *flatten* · *mktree* with *fun* using the `fusion` theorem. The LISTLENIND tactic is then used to prove the resulting equality by complete induction on the length of L. A minor complication is introduced by the fact that the FOLD tactic does not work for certain abstractions,[3] which forces us to work with the unfolded terms in some places. The proof is about 31 steps long.

```
* THM quicksort_by_fusion
∀T:𝕌. ∀≤:T → T → 𝔹. ∀L:T List.
    quicksort(≤,L) = flatten(mktree(≤)(L))
```

Fig. 30. Theorem `quicksort_by_fusion`

10 A Formal Correctness Proof

QUICKSORT is a sorting algorithm: For every list L, it should return an ordered permutation of that list. We prove that QUICKSORT is correct by first proving that it returns an ordered list, and secondly by proving that it returns a permutation of its input. The first proof is based on the representation of *quicksort* as *flatten* · *mktree*, while the second proof uses the definition of *quicksort* directly.

10.1 Quicksort Returns an Ordered List

We say a list L is *ordered* if the elements in L are in ascending order (with respect to a relation $\leq$).

Definition 13 (ordered) *Suppose T is a type and $\leq : T \times T \to \mathbb{B}$. Define $ordered(\leq, L) \in \mathbb{B}$ recursively by*

$$ordered(\leq, L) = \begin{cases} true & \text{if } L = [] \\ (\forall x \in t.\ h \leq x) \wedge ordered(\leq, t) & \text{if } L = h :: t \end{cases}.$$

By checking whether the head of the list is below *every* other element in the list (instead of just checking whether it is below the second element), we avoid having to check if there exists a second element in the list. The NUPRL abstraction defining `ordered` is shown in Figure 31. The well-formedness theorem `ordered_wf` proves $ordered(\leq, L) \in \mathbb{B}$ if T is a type, $\leq : T \times T \to \mathbb{B}$ and $L \in List(T)$. The well-formedness theorem is proved by structural induction on L using the LISTIND tactic.

```
* ABS ordered
ordered(≤,L) ==
  (letrec recfun(L) =
      case L of
        [] => tt
      | h::t => ∀x∈₂t.(h ≤ x) ∧_b recfun t esac )
  L
```

Fig. 31. Abstraction `ordered`

[3] Folding abstractions that contain `so_apply` seems to be a problem in some cases.

To prove that the list returned by *quicksort* = *flatten* · *mktree* is ordered, we first prove that *mktree* creates an ordered tree. Before we can define what it means for a binary tree to be ordered, we need to define a function that computes whether some predicate $P[x]$ holds for every element x in a tree. The abstraction defining `tree_all_2` is shown in Figure 32. The name of the function ends with '`_2`' to indicate that a boolean value is returned (as opposed to a proposition in $\mathbb{P}$), thereby following the naming scheme for the `list_all` functions defined in the `LIST_3` library.

```
* ABS tree_all_2
∀x∈₂B.P[x] ==
  (letrec recfun(B) =
      case B of
        inl(y) => tt
      | inr(z) => let t,B1,B2 = z in P[t] ∧_b recfun B1 ∧_b recfun B2 )
  B
```

Fig. 32. Abstraction `tree_all_2`

The well-formedness theorem `tree_all_2_wf` shows that $(\forall x \in_2 B.P[x])$ is a boolean value for every type T, $P : T \to \mathbb{B}$, and $B \in \mathit{BinTree}(T)$. It is proved in about eight steps; we use the RecElimination tactic in its proof for structural induction on B. We can now define when a binary tree is ordered.

Definition 14 (treeordered) *Suppose T is a type and $\leq : T \times T \to \mathbb{B}$. Define $\mathit{ordered}(\leq, B) \in \mathbb{B}$ recursively by*

$$\mathit{ordered}(\leq, B) = \begin{cases} \mathit{true} & \text{if } B = \mathit{leaf} \\ (\forall z \in B_1.\, z \leq t) \wedge (\forall z \in B_2.\, \neg(z \leq t)) & \text{if } B = \mathit{node}(t, B_1, B_2) \\ \quad \wedge\, \mathit{ordered}(\leq, B_1) \wedge \mathit{ordered}(\leq, B_2) & \end{cases} .$$

The corresponding Nuprl abstraction `treeordered` is shown in Figure 33. As usual, we prove a well-formedness theorem for it: `treeordered_wf` just shows that for every type T, $\leq : T \times T \to \mathbb{B}$, and $B \in \mathit{BinTree}(T)$, $\mathit{ordered}(\leq, B) \in \mathbb{B}$. It is proved in about six steps by structural induction on B.

```
* ABS treeordered
ordered(≤,B) ==
  (letrec recfun(B) =
      case B of
        inl(x) => tt
        | inr(y) => let t,B1,B2 = y in
                      ∀z∈₂B1.(z ≤ t)
                      ∧_b ∀z∈₂B2.(¬_b(z ≤ t))
                      ∧_b recfun B1
                      ∧_b recfun B2 )
  B
```

Fig. 33. Abstraction `treeordered`

Lemma 7. *Suppose T is a type and $\leq : T \times T \to \mathbb{B}$. Then*

$$\mathit{ordered}(\leq, \mathit{mktree}(\leq)(L))$$

for all $L \in List(T)$.

Figure 34 shows the NUPRL theorem `ordered_mktree`. The arrow '↑' (*assert*) is used to turn the boolean value $ordered(\leq, mktree(\leq)(L))$ into a proposition, i.e. `tt` becomes *True*, `ff` becomes *False*.

```
* THM ordered_mktree
∀T:𝕌. ∀≤:T → T → 𝔹. ∀L:T List. ↑ordered(≤,mktree(≤)(L))
```

Fig. 34. Theorem `ordered_mktree`

To prove the formal theorem, we need three lemmata: Firstly, that $f[x]$ holds for all x in $filter(f; L)$ assuming T is a type, $f : T \to \mathbb{B}$ and $L \in List(T)$. Secondly, that $P[x]$ holds for all $x \in L$ if and only if $P[x]$ holds for all x in $filter(f; L)$ and for all x in $filter(\neg f; L)$ assuming T is a type, $P, f : T \to \mathbb{B}$, and $L \in List(T)$. Finally, that $f[x]$ holds for all x in L if and only if $f[x]$ holds for all x in $mktree(\leq)(L)$ assuming T is a type, $\leq : T \times T \to \mathbb{B}$, $f : T \to \mathbb{B}$, and $L \in List(T)$. The lemmata are shown in Figures 35, 36, and 37 respectively.

```
* THM filter_all_2
∀T:𝕌. ∀f:T → 𝔹. ∀L:T List. ↑∀x∈₂filter(f;L).f[x]
```

Fig. 35. Theorem `filter_all_2`

The `filter_all_2` lemma is proved in about eight steps by structural induction on L using the LISTIND tactic. The base case is proved in a single step by the AUTO tactic. For the case $L = u :: v$, the IFTHENELSE-CASES tactic is used to do a case split on $f[u]$.

```
* THM list_all_2_filter_filter
∀T:𝕌. ∀f,P:T → 𝔹. ∀L:T List.
   ∀x∈₂L.P[x] = ∀x∈₂filter(f;L).P[x] ∧_b ∀x∈₂filter((λz.¬_bf[z]);L).P[x]
```

Fig. 36. Theorem `list_all_2_filter_filter`

The `list_all_2_filter_filter` lemma is also proved by structural induction on L. The case $L = []$ is proved in a single step again, and for the case $L = u :: v$, we do a case split on $f[u]$ by IFTHENELSECASES. The resulting equalities are proved using the associativity and commutativity of $\wedge_b$. The proof is about eleven steps long.

```
* THM mktree_all_2
∀T:𝕌. ∀≤:T → T → 𝔹. ∀f:T → 𝔹. ∀L:T List. ∀x∈₂L.f[x] = ∀x∈₂mktree(≤)(L).f[x]
```

Fig. 37. Theorem `mktree_all_2`

Proving the `mktree_all_2` lemma is slightly more complicated. We start by using the LISTLENIND tactic for complete induction on the length of L, followed by the LISTIND tactic to differentiate between the two cases $L = []$ and $L = u :: v$. For the base case, we instantiate the lemma mktree_of_nil, and for the case $L = u :: v$, we use the mktree_of_cons lemma. The induction hypothesis is then used on the two lists $\mathit{filter}(\cdot\ \mathit{below}(\leq)\ u; v)$ and $\mathit{filter}(\cdot\ \mathit{above}(\leq)\ u; v)$. Finally the list_all_2_filter_filter lemma is used to prove the equivalence of $(\forall x \in_2 v.f[x]))$ and $(\forall x \in_2 \mathit{filter}(\cdot\ \mathit{below}(\leq)\ u; v).f[x]) \wedge (\forall x \in_2 \mathit{filter}(\cdot\ \mathit{above}(\leq)\ u; v).f[x])$. The proof is about 23 steps long.

The proof of ordered_mktree then requires about 26 steps. It is based on complete induction on the length of L, using the ListLenInd tactic followed by ListInd. About 20 of those steps are needed to prove the case $L = u :: v$.

Our next step in proving that *quicksort* returns an ordered list is to show that $\mathit{flatten}(B)$ is an ordered list if B is an ordered tree.

Lemma 8. *Suppose T is a type, $\leq : T \times T \to \mathbb{B}$ is transitive and total (i.e. $x \leq y$ or $y \leq x$ for all $x, y \in T$), and $B \in \mathit{BinTree}(T)$. Then*

$$\mathit{ordered}(\leq, B) \Rightarrow \mathit{ordered}(\leq, \mathit{flatten}(B)).$$

The corresponding Nuprl theorem ordered_flatten is shown in Figure 38.

```
* THM ordered_flatten
∀T:𝕌.
 ∀≤:{≤:T → T → 𝔹| Trans(T;x,y.↑≤[x;y]) ∧ Connex(T;x,y.↑≤[x;y])} .
  ∀B:BinTree(T).
   ↑ordered(≤,B) ⇒ ↑ordered(≤,flatten(B))
```

Fig. 38. Theorem ordered_flatten

We need a number of fairly self-evident lemmata before we can formally prove this theorem. The list_all_2_append_lemma lemma shown in Figure 39 proves that a property $P[x]$ holds for all x in $L@M$ if and only if it holds for all x in L and for all x in M. In other words, '∀' distributes over *append*. Using the ListInd tactic for structural induction on L, the lemma is proved in about four steps.

```
* THM list_all_2_append_lemma
∀T:𝕌. ∀P:T → 𝔹. ∀L,M:T List.
 ∀x∈₂(L @ M).P[x] = ∀x∈₂L.P[x] ∧_b ∀x∈₂M.P[x]
```

Fig. 39. Theorem list_all_2_append_lemma

Figure 40 shows a lemma proving that a list of the form $L@(t :: M)$ is ordered if and only if L is ordered, M is ordered, $x \leq t$ for all x in L, and $t \leq x$ for all x in M. To prove the lemma, we use structural induction on L, the list_all_2_append_lemma lemma and a number of other lemmata. A nested induction on M and several case splits are required for the case where $L = u :: v$. The proof is about 60 steps long.

The flatten_all_2 lemma (see Figure 41) shows that a property $f[x]$ holds for all x in a binary tree B if and only if it holds for all x in $\mathit{flatten}(B)$. This lemma is similar to the mktree_all_2 lemma proved earlier. The proof is by structural induction on B. It requires about 29 steps, including one instantiation of the list_all_2_append_lemma lemma.

Figure 42 shows another lemma that we need, list_all_2_implies_lemma. It proves that if $P[x]$ and $(P[x] \Rightarrow Q[x])$ hold for all x in a list L, then $Q[x]$ holds for all x in L. The lemma is proved in about 13

```
* THM ordered_append
∀T:𝕌. ∀≤:{≤:T → T → 𝔹| Trans(T;x,y.↑≤[x;y])} . ∀L,M:T List. ∀t:T.
 ordered(≤,L @ (t::M)) =
   ∀x∈₂L.(x ≤ t) ∧b ∀x∈₂M.(t ≤ x) ∧b ordered(≤,L) ∧b ordered(≤,M)
```

Fig. 40. Theorem `ordered_append`

```
* THM flatten_all_2
∀T:𝕌. ∀f:T → 𝔹. ∀B:BinTree(T). ∀x∈₂B.f[x] = ∀x∈₂flatten(B).f[x]
```

Fig. 41. Theorem `flatten_all_2`

```
* THM list_all_2_implies_lemma
∀T:𝕌. ∀P,Q:T → 𝔹. ∀L:T List.
 ↑∀x∈₂L.P[x] ∧ ↑∀x∈₂L.(P[x] ⇒b Q[x]) ⇒ ↑∀x∈₂L.Q[x]
```

Fig. 42. Theorem `list_all_2_implies_lemma`

steps by structural induction on L; many of those steps just deal with the fairly technical difference between boolean values and propositions.

Our last lemma for now is shown in Figure 43. The `list_all_2_if_all` lemma proves that a property $P[x]$ holds for all x in a list $L \in \mathit{List}(T)$ if it holds for all $x \in T$. It is proved in about six steps by structural induction on L.

```
* THM list_all_2_if_all
∀T:𝕌. ∀P:T → 𝔹. ∀L:T List. (∀x:T. ↑P[x]) ⇒ ↑∀x∈₂L.P[x]
```

Fig. 43. Theorem `list_all_2_if_all`

Given these lemmata, the proof of `ordered_flatten` requires about 58 steps. The RecElimination tactic is used for structural induction on B. The base case is then proved in about six steps simply by unfolding definitions. Proving the case $B = \mathit{node}(t, B_1, B_2)$ requires the use of the lemmata `ordered_append`, `flatten_all_2`, `list_all_2_implies_lemma` and `list_all_2_if_all`.

We proved that *mktree* always creates an ordered tree, and that *flatten* flattens an ordered tree into an ordered list. Given the `quicksort_by_fusion` theorem from Section 9, the proof that Quicksort always returns an ordered list is quite simple now.

```
* THM ordered_quicksort
∀T:𝕌.
 ∀≤:{≤:T → T → 𝔹| Trans(T;x,y.↑≤[x;y]) ∧ Connex(T;x,y.↑≤[x;y])} .
  ∀L:T List.
   ↑ordered(≤,quicksort(≤,L))
```

Fig. 44. Theorem `ordered_quicksort`

To prove the `ordered_quicksort` theorem shown in Figure 44, we first replace *quicksort*$(\leq, L)$ with *flatten*(*mktree*$(\leq)(L)$) using the `quicksort_by_fusion` theorem. After using the `ordered_flatten` lemma then, we only have to prove that *mktree*$(\leq)(L)$ is ordered. This is proved by the `ordered_mktree` lemma. All well-formedness goals are discharged by NUPRL's AUTO tactic, so the whole proof requires only three steps.

10.2 Quicksort Returns a Permutation of its Input

In the previous subsection we proved that QUICKSORT always returns an ordered list. To prove that QUICKSORT is a sorting algorithm, it remains to show that the list returned by QUICKSORT is a permutation of the input list.

Theorem 15. *Suppose T is a type, $eq : T \times T \to \mathbb{B}$ is a function with $eq(x, y) = true$ if and only if $x = y$ for all $x, y \in T$ (in other words, equality in T is decidable), and $\leq : T \times T \to \mathbb{B}$. Furthermore, suppose $x \in T$ and $L \in List(T)$. Then x occurs in quicksort$(\leq, L)$ exactly as often as in L.*

The idea of counting the occurrences of an element in L and in *quicksort*$(\leq, L)$ is borrowed from [7]. Figure 45 shows the NUPRL theorem `list_count_quicksort`. We used the abstractions `discrete_equality`, which can be found in the `DISCRETE` library, and `list_count` from the `LIST_3` library to state the theorem. We need a decidable equality on T in order to be able to count the occurrences of a given element $x \in T$ in the two lists L and *quicksort*$(\leq, L)$: If we could not tell whether two elements $x, y \in T$ are equal, we could not compare x to the elements in L and *quicksort*$(\leq, L)$.

```
* THM list_count_quicksort
∀T:𝕌. ∀eq:{T=₂}. ∀≤:T → T → 𝔹. ∀L:T List. ∀x:T.
  |x∈quicksort(≤,L)| = |x∈L|
```

Fig. 45. Theorem `list_count_quicksort`

We do not prove this theorem directly. Instead, we prove three lemmata first. The first lemma, `list_count_over_filter_l` is shown in Figure 46. It proves that an element x occurs in the list *filter*$(f; L)$ exactly as often as in L if $f[x]$ is true, and zero times otherwise. The lemma is proved in about 33 steps using the LISTIND tactic for structural induction on L, combined with several applications of the IFTHENELSECASES tactic for case splits on $f[x]$ and—in the case $L = u :: v$—on $f[u]$. The fact that we can decide whether x is equal to u (via the *eq* function) is crucial to the proof.

```
* THM list_count_over_filter_lemma
∀T:𝕌. ∀eq:{T=₂}. ∀f:T → 𝔹. ∀L:T List. ∀x:T.
  |x∈filter(f;L)| = if f[x] then |x∈L| else 0 fi
```

Fig. 46. Theorem `list_count_over_filter_lemma`

The second lemma, shown in Figure 47, states that an element x occurs in L exactly as often as in the two lists *filter*$(f; L)$ and *filter*$(\neg f; L)$ together. It is proved in about 16 steps. We apply the `list_count_over_filter_lemma` lemma twice in its proof: first to the list *filter*$(f; L)$, and then to the list *filter*$(\neg f; L)$.

Figure 48 shows the third lemma. This lemma is an instance of `list_count_over_filter_lemma` that has been specialized by the predicates *below* and *above*. The lemma is trivially proved by making the instantiation.

```
* THM list_count_filter_filter_lemma
∀T:𝕌. ∀eq:{T=₂}. ∀f:T → 𝔹. ∀L:T List. ∀x:T.
  |x∈filter(f;L)| + |x∈filter((λz.¬bf[z]);L)| = |x∈L|
```

Fig. 47. Theorem `list_count_filter_filter_lemma`

```
* THM list_count_below_above
∀T:𝕌. ∀eq:{T=₂}. ∀≤:T → T → 𝔹. ∀L:T List. ∀u,x:T.
  |x∈filter((λb.b below(≤) u);L)| + |x∈filter((λb.b above(≤) u);L)| = |x∈L|
```

Fig. 48. Theorem `list_count_below_above`

The proof of `list_count_quicksort` now requires about 55 steps. The ListLenInd tactic is used for complete induction on the length of L, followed by the ListInd tactic two differentiate between the two possible cases $L = []$ and $L = u :: v$. The case $L = []$ is proved in a single step by the Auto tactic after unfolding the definition of *quicksort*. For the case $L = u :: v$, we apply the `list_count_over_append_lemma` lemma from the `LIST_3` library to the two lists $\mathit{quicksort}(\leq, \mathit{filter}(\cdot\ \mathit{below}(\leq)\ u; v))$ and $u :: \mathit{quicksort}(\leq, \mathit{filter}(\cdot\ \mathit{above}(\leq)\ u; v))$. The induction hypothesis is then applied to the lists $\mathit{quicksort}(\leq, \mathit{filter}(\cdot\ \mathit{below}(\leq)\ u; v))$ and $\mathit{quicksort}(\leq, \mathit{filter}(\cdot\ \mathit{above}(\leq)\ u; v))$. Finally `list_count_below_above` is used on the two lists $\mathit{filter}(\cdot\ \mathit{below}(\leq)\ u; v)$ and $\mathit{filter}(\cdot\ \mathit{above}(\leq)\ u; v)$.

This does not only complete the proof that Quicksort returns a permutation of its input list, but it is also the last step in our correctness proof for Quicksort. The next section presents an alternative approach to proving that Quicksort returns a permutation of its input.

10.3 Quicksort Returns a Permutation of its Input: A Second Proof

To prove that Quicksort returns a permutation of its input in the previous section, we counted the number of occurrences of elements in the lists L and $\mathit{quicksort}(\leq, L)$. We cannot do this unless equality on T is decidable. This is not a real restriction if $\leq$ is a decidable order relation on T: Then $x = y \iff (x \leq y \wedge y \leq x)$ for all x and y in T.[4] However, all theorems that we proved in the previous section only required $\leq$ to be total (i.e. $x \leq y \vee y \leq x$ for all $x, y \in T$) and transitive (i.e. $(x \leq y \wedge y \leq z) \Rightarrow x \leq z$ for all $x, y, z \in T$), and there is a different approach to proving that Quicksort returns a permutation of its input—an approach that does not require equality on T to be decidable.

This approach is based on the inductive definition of `permutation` shown in Figure 49. The definition can be found in the `LIST_3` library.

We also need two self-evident lemmata: that *permutation* is transitive, and that *permutation* distributes over append. The former is shown in Figure 50, and the latter in Figure 51.

We now prove a lemma similar to the `list_count_filter_filter_lemma` lemma shown in Figure 47: L is a permutation of $\mathit{filter}(f; L) @ \mathit{filter}(\neg f; L)$. This lemma, which is shown in Figure 52, is proved in about 23 steps by structural induction on L.

The `permutation_below_above` lemma (see Figure 53) simply results from applying the `permutation_filter_filter_lemma` lemma to the two predicates *below* and *above*. It is proved in about three steps.

We can now show that $\mathit{quicksort}(\leq, L)$ is a permutation of L. Figure 54 shows the corresponding Nuprl theorem. It is proved by complete induction on the length of L using the ListLenInd tactic, followed by the ListInd tactic to differentiate between $L = []$ and $L = u :: v$. The case $L = []$ is then proved in a single step

[4] The '⇒' direction follows from the reflexivity of $\leq$, and the antisymmetry of $\leq$ implies the '⇐' direction.

```
* ABS permutation
perm(L,M) ==
  (letrec perm(L)(M) =
      case L of
        [] => case M of
                [] => True
              | h::t => False
              esac
      | h::t => case M of
                  [] => False
                | h'::t' => ∃N,N':T List. M = N @ (h::N') ∧ perm t (N @ N')
                esac
      esac )
      L
  M
```

Fig. 49. Abstraction `permutation`

```
* THM permutation_transitive
∀T:𝕌. ∀L,M,N:T List. perm(L,M) ⇒ perm(M,N) ⇒ perm(L,N)
```

Fig. 50. Theorem `permutation_transitive`

```
* THM permutation_over_append_lemma
∀T:𝕌. ∀A,B,X,Y:T List. perm(A,X) ∧ perm(B,Y) ⇒ perm(A @ B,X @ Y)
```

Fig. 51. Theorem `permutation_over_append_lemma`

```
* THM permutation_filter_filter_lemma
∀T:𝕌. ∀f:T → 𝔹. ∀L:T List.
 perm(L,filter(f;L) @ filter((λz.¬_b f[z]);L))
```

Fig. 52. Theorem `permutation_filter_filter_lemma`

```
* THM permutation_below_above
∀T:𝕌. ∀≤:T → T → 𝔹. ∀L:T List. ∀u:T.
 perm(L,filter((λb.b below(≤) u);L) @ filter((λb.b above(≤) u);L))
```

Fig. 53. Theorem `permutation_below_above`

by unfolding definitions and the AUTO tactic. Proving the case $L = u :: v$ requires approximately 34 steps. A number of lemmata are instantiated in this part of the proof. Altogether, the proof is about 39 steps long.

```
* THM permutation_quicksort
∀T:𝕌. ∀≤:T → T → 𝔹. ∀L:T List. perm(L,quicksort(≤,L))
```

Fig. 54. Theorem `permutation_quicksort`

This completes our second proof that QUICKSORT returns a permutation of its input.

References

1. Richard S. Bird. Functional algorithm design. In Bernhard Moller, editor, *Mathematics of Program Construction '95*, volume 947 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, 1995.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.
3. M. Davis. Deforestation: Transformation of functional programs to eliminate intermediate trees. Master's thesis, Oxford University, 1987.
4. William F. Eddy and Mark J. Schervish. How many comparisons does Quicksort use? *Journal of Algorithms*, 19(3):402–431, November 1995.
5. A. Gill, Launchbury J., and Peyton Jones S.L. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.
6. C. A. R. Hoare. ACM Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, July 1961.
7. Natarajan Shankar. Steps toward mechanizing program transformations using PVS. *Science of Computer Programming*, 26(1-3):33–57, 1996.
8. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88. European Symposium on Programming, Nancy, France, 1988*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Berlin: Springer-Verlag, 1988.