Balanced Trees Inhabiting Functional Parallel Programming

Zhili Pan Samuel Tepoorten

Formal Verification 2024-2025 Final Report

1 Introduction

The existing data structures provided in functional programming languages often face limitations regarding parallel execution for operations such as summation. These operations are typically performed in an inefficient sequential manner. Morihata and Matsuzaki (2011) proposed using balanced trees as abstract data structures to support divide-and-conquer parallelism and provided sample data structure definitions. However, they focused on systematical reasoning, thus falling short of providing detailed implementations. This project seeks to address this gap by developing and demonstrating practical implementations that support and validate the theoretical concepts outlined in their paper.

2 Theoretical Background

In our project, we developed recursive functions that support efficient parallel computation for both lists and binary trees. We explored how these structures can be implemented and verified using Stainless. Additionally, we delved into both basic and advanced structures to ensure a more efficient approach to leveraging these methods and functions.

Whenever we would like to support divide and conquer strategies and enable parallelism we would like to use balanced Trees. In doing this we desire to implement recursive parallel functions. Here is where join lists comes into play. But what is a Join List exactly? As we will see s Join list is a binary tree representation of a list, where each node either contains a single element or combines two sublists. This structure supports efficient divide-and-conquer operations, making it well-suited for parallel processing tasks;

- **Single** represents a join list containing a single element.
- **Join** combines two `JList` structures into one, facilitating the creation of binary tree structures for parallel processing.

Definition:

$$\textbf{data JList}\,\alpha = \textbf{Single}\,\alpha \mid \textbf{Join}\,(\textbf{JList}\,\alpha)\,(\textbf{JList}\,\alpha)$$

Conversion: From Join List back to sequential list:

$$\begin{aligned}&j2l :: \forall\alpha.\,\textbf{JList}\,\alpha \to [\alpha]\\ &j2l\,(\textbf{Single}\,a) = [a]\\ &j2l\,(\textbf{Join}\,l\,r) = j2l\,(l) +\!\!+\, j2l\,(r)\end{aligned}$$

In order to support efficient divide and conquer parallel operation, and ensure a consistent height on the tree (log(n)) we would like to balance our join Lists. In this regard, we concurrently implemented a balanced version of join lists, using self-balanced binary tree to maintain balance during modifications.

Another parallel optimization comes by introducing the notion of *Shunt Trees*, more specifically, when delving into parallel reductions, here is how a Shunt Contraction Scheme enables efficient and parallel processing of tree structures:

A basic binary tree consists of:

- *Leaves (Tip)*: Values A
- *Internal nodes (Bin)*: Connect two subtrees and hold values

Definition:

$$\text{data } \mathit{Tree}\ \alpha\ \beta = \mathit{Tip}\ \alpha \mid \mathit{Bin}\ (\mathit{Tree}\ \alpha\ \beta)\ \beta\ (\mathit{Tree}\ \alpha\ \beta)$$

The challenge of basic Trees is that Unbalanced (list-like) trees require O(h) steps, where h is the height. Also, parallelizing operations on trees requires reducing the tree depth. Here is where the Shunt and the Tree contraction concept comes into play: A Shunt operation change the shape of the tree, it performs the following: it first Removes a leaf and its parent node and then onnects the subtree directly to its grandparent node:

Definition:

$$\text{shunt } \forall\alpha,\beta : Tree\ \alpha\ \beta \rightarrow Tree\ \alpha\ \beta$$

$$\text{shunt}(\text{Bin } t\_(\text{Tip}\_)) = t \quad \text{and} \quad \text{shunt}(\text{Bin}(\text{Tip}\_)\_t) = t$$

The *scs* function generalize the shunt operation allowing us to:

- Process leaves using $A \rightarrow \alpha$.
- Process internal nodes using $B \rightarrow \beta$.
- Combine intermediate results with Shunt $\alpha\,\beta$.

Definition:

$$\text{scs} :: \forall\alpha,\beta.(A \rightarrow \alpha) \rightarrow (B \rightarrow \beta) \rightarrow \text{Shunt } \alpha\beta \rightarrow \text{Tree } AB \rightarrow \alpha$$

In order to make sure that the tree is reconstructed correctly during the Shunt contraction, we use two key concepts:

- **Hole**: Creates a context with a "hole" that represents a missing subtree (a Tree with a hole)

  Definition:

$$\text{hole} :: \forall\alpha,\beta.\beta \rightarrow \text{Context } \alpha\beta$$

$$\text{hole } a = \lambda(l,r) \rightarrow \text{Bin } l\ a\ r$$

- **Connect**: Combines a context with subtrees to reconstruct the tree.

$$\text{connect} :: \forall\alpha,\beta.\text{Shunt } (\text{Tree } \alpha\beta)\ (\text{Context } \alpha\beta)$$

$$\text{connect} = (\psi_L, \psi_R, \psi_N)$$

Now with this background we can construct our Shunt Trees which they both reduce the Tree depth and support parallelization of operations:

Definition:

$$\begin{aligned}
\textbf{data ST}\ \alpha\,\beta &= \textbf{T}\,\alpha \\
&\mid\ \textbf{N}\,(\textbf{ST}\ \alpha\,\beta)\ (\textbf{ST}_\bullet\,\alpha\,\beta)\ (\textbf{ST}\ \alpha\,\beta) \\
\textbf{data ST}_\bullet\,\alpha\,\beta &= \textbf{H}_\bullet\,\beta \\
&\mid\ \textbf{L}_\bullet\,(\textbf{ST}_\bullet\,\alpha\,\beta)\ (\textbf{ST}_\bullet\,\alpha\,\beta)\ (\textbf{ST}\ \alpha\,\beta) \\
&\mid\ \textbf{R}_\bullet(\textbf{ST}\ \alpha\,\beta)\ (\textbf{ST}_\bullet\,\alpha\,\beta)\ (\textbf{ST}_\bullet\,\alpha\,\beta)
\end{aligned}$$

With ST· for Context, ST for binary tree.

Similarly to join Lists, balancing is necessary to necessary to reduce tree height and fully exploit parallelism. The contraction algorithm reduces tree height to *O(logn).*

3 Implementation

3.1 Preliminaries

We implemented the proposed data structures in Scala 3 and verified them using Stainless. The project code is in our Github repository (https://github.com/sasoripathos/ShuntTree). The code is organized as follows:

- `src/Helpers.scala`: This file includes various helper functions and proofs for lemmas that support proofs in other files. Additionally, it defines shared objects and data types utilized in proofs across other files.
- `src/JoinList.scala`: This file provides the implementation and proofs for an unbalanced version of JoinList.
- `src/BalancedJoinList.scala`: This file provides the implementation and proofs for a balanced version of JoinList.
- `src/Tree.scala`: This file provides an implementation and proof for the proposed Tree data structure.
- `src/ShuntTree.scala`: This file contains an implementation for the proposed ShuntTree data structure.

3.2 JoinList

3.2.1 Unbalanced JoinList

Morihata and Matsuzaki (2011) ignored the empty list case in their paper for simplicity. We decided to include it in our implementation to support general use cases. The original definition from the paper implies that for ***Join*** x y, both sublists x and y are non-empty. Hence, our definition is as follows. We used the Stainless *require()* function to ensure the sublist property during the consturction.

```scala
sealed abstract class JoinList[T]
case class Empty[T]() extends JoinList[T]
case class Single[T](value: T) extends JoinList[T]
case class Join[T](left: JoinList[T], right: JoinList[T]) extends JoinList[T] {
  require(
    left != Empty[T]() && right != Empty[T]()
  )
}
```

The authors proposed JoinList as a replacement for the existing list, so in this project, we tried to implement our JoinList with a comparable functionality as supported in Stainless's *List[T]* data structure. Namely, we successfully implemented the methods listed in Appendix A.

Our main strategy to prove the correctness of our implementation was to show the computation result of our method being the same as that of the corresponding Stainless List method. Thus, most of our code describes the postconditions as the following pseudocode, where the toList method is where we implemented the j2l conversion.

```scala
def methodA(...): = {
    ......
}.ensuring(_ == jl.toList.methodA(...))
```

For preconditions, we followed the same practices as in the Stainless List. In the process, We utilized some lemmas from Stainless ListSpec. We also proved some lemmas for Stainless List in Helper.scala to facilitate our proofs. For example, one lemma is for foldLeft on a concatenated list.

```
def listFoldLeftCombine[T, R](l1: List[T], l2: List[T], f: (R, T) => R, basecase: R):
↪  Unit = {
    ...... check details on GitHub repository ......
  }.ensuring((l1 ++ l2).foldLeft(basecase)(f) ==
  ↪  l2.foldLeft(l1.foldLeft(basecase)(f))(f))
```

In this version, we don't care about balancing the JoinList. Hence the operations that add or remove elements are very straightforward. For example, the list concatenation (++) is just creating a new ***Join*** x y when both x and y are non-empty JoinList.

3.2.2 Balanced JoinList

Morihata and Matsuzaki (2011) proposed to implement JoinList by self-balancing binary search trees, like biased search trees, to ensure worst-case complexity. Since we don't have previous knowledge about the biased search tree, we chose other simpler tree balancing techniques in our implementation. It is clear that any self-balancing binary tree concepts are sufficient here.

In the balanced version of JoinList, we added a height method and used subtrees' height difference to determine the balance. The type definition is similar to the one in the unbalanced version, with only one extra constrain for balancing in constructing ***Join*** x y: the height differences between 2 sublists should be within ($\leq$) 1.

```
  // ...... other case are same as above ......
  case class Join[T](left: JoinList[T], right: JoinList[T]) extends JoinList[T] {
    require(
      left != Empty[T]() && right != Empty[T]()
      && BigInt(-1) <= left.height - right.height && left.height - right.height <=
      ↪  BigInt(1)
    )
  }
```

This constraint in the constructor precludes the creation of an unbalanced tree. We have proof of this.

```
  extension[T](jl: JoinList[T]) {
    def isBalanced: Boolean = {
      jl match {
        case Join(l, r) => {
          l.isBalanced && r.isBalanced && BigInt(-1) <= l.height - r.height &&
          ↪  l.height - r.height <= BigInt(1)
        }
        case _ => true // true for empty and single
      }
    }.ensuring(_ == true)
  }
```

In the balanced JoinList, most implemenations of list methods (i.e. apply) remain the same as those in the unbalanced Joinlist. We have to reimplement the methods which add or remove list elements. In the unbalanced version, adding or removing a single element from a Join list with more than 1 element would only affect one of its sublist. We could directly "join" the modified sublist with the untouched one. In balanced version, we need to find a way to combine the balanced mordified sublist with the balanced untouched sublist. Hence we need to have a special way to concatenate (++) 2 balanced JoinLists.

The main logic for (++) in the balanced version is: given 2 (balanced) JoinList x and y

1. **Lemma:** By Join, the new balanced join list would no shorter than x and y, and would at most grow by 1 in height.
2. If the height difference between x and y are within (≤) 1, directly construct ***Join*** *x y*.
3. Otherwise,
   - If y is taller, we want to combine x with y's left child to preserve the relative order. Specifically,
     - If y's right child is not shorter than y's left child, based on the lemma, we can simply recursively combine x and y's left child and construct a new JoinList with the result and y's right child.
     - Otherwise, recursively combine x with y's left grandchild. Construct new JoinList depending on the height of result tree.
   - otherwise, similar as above but swap left and right, want to combine y with x's right subtree.

Specifically, the postcondition for (++) in the balanced version is as follows, where *jl* is *x*, *other* is *y*.

```
.ensuring( res => (
    // for correctness
    res.toList == jl.toList ++ other.toList
    && res.isBalanced // this we proved always true as above
    // Below are for the Lemma
    && res.height <= max(jl.height, other.height) + 1
    && res.height >= max(jl.height, other.height)
  ))
```

With this balanced version of (++), other operations that change the number of list elements (i.e. : +, ::) can be reduced to (++. The proofs were seamless.

3.2.3 List Aggregation

With the JoinList, we theoretically can achieve all parallel execution as long as the aggregation operation is associative. In the paper, Morihata and Matsuzaki (2011) formally describe a (associative) list aggregation function as:

$$h :: \forall\beta.(\beta \to \beta \to \beta) \to (A \to \beta) \to [A] \to \beta$$

We realized this computation skeleton in Stainless as follows. We proved the correctness of all possible associative list aggregation functions by showing they produce the same results on our JoinList and sequential list. Limited by our knowledge and capabilities, we didn't prove the time complexity of our implementation in a possible parallel execution environment.

```
abstract class ListAggFunction[T]:
  def execute(x: T, y: T): T
  @law
  def isAssociative(x: T, y: T, z: T): Boolean = {
    execute(execute(x, y), z) == execute(x, execute(y, z))
  }
```

3.3 Tree

Based on the method described above, Morihata and Matsuzaki (2011) explain how it scales up to binary trees, that we defined as:

```
sealed abstract class Tree[A, B]
case class Tip[A, B](value: A) extends Tree[A, B]
case class Bin[A, B](value: B, left: Tree[A, B], right: Tree[A, B]) extends Tree[A,
↪  B]
```

By this definition, each node in a tree is either a leaf node (i.e. with no child) or an internal node (with two children). This forces such tree always have odd number of nodes (as proved in our code). we acknowledged this definition is too limited to support simple tree operations like single node insertion. Considering the scope of this project, we decided to leave the generalization of this Tree type as potential future work.

We implemented some basic tree operations. One example was to find a specific node in this tree. We made a method apply, that finds a node based on its order in tree preorder traversal. Since a Tree has potential different types on leaf and internal nodes, we decided to use Scala's Either type to hold the result. The correctness proof for such methods was similar to what we did in JoinList, reducing the tree to a List[Either [A, B]].

```
extension[A, B](tr: Tree[A, B]) {
  def toInOrderList: List[Either[A, B]] = {
    .......
  }.ensuring(res => res.head.isLeft && res.last.isLeft)

  def apply(i: BigInt): Either[A, B] = {
    ......
  }.ensuring(_ == tr.toInOrderList.apply(i))
}
```

For parallel aggregation functions on Tree, it is achievable in a similar fashion as in JoinList. Morihata and Matsuzaki (2011) mentioned one computation skeleton for map on Tree

$$map :: \forall \alpha\ \beta\ \alpha'\ \beta'.(\alpha \to \alpha') \to (\beta \to \beta') \to Tree\ \alpha\ \beta \to Tree\ \alpha'\ \beta'$$

We implemented and proved it by showing it is equivalent to *map* over a sequential List. During the process, we defined a customized *map* function for Scala Either (called *caseMap*) to better serve our purpose.

```
def map[C, D](lf: A => C, rf: B => D): Tree[C, D] = {
   ......
  }.ensuring(res => (
      res.size == tr.size
      &&
      res.toInOrderList == tr.toInOrderList.caseMap(lf, rf)
    )
  )
```

3.4 ShuntTree

Functions that aggregate information on trees have a similar behaviour as those on the list. They would compute on nodes and merge the result from a node with the results from both its subtrees. If one considers aggregation as replacing a subtree with a single node that "holds" the aggregation result of the subtree, then such aggregations are essentially reducing a tree to a leaf, while doing computations during the process. This matches the definition of the parallel tree contraction algorithm stated by Reif (1993). Hence, one can implement aggregation functions via a parallel tree contraction algorithm. An example of a tree contraction algorithm is Shunt, which operates on leaves (Reif, 1993). Since Shunt operations aggregate information from three nodes, Morihata and Matsuzaki (2011) proposed a record type to express aggregating operations in three cases. In our project we defined the Shunt operation as follows:

```
  sealed trait ShuntOperation[X, Y]:
    def left(b: Y, a: Y, c: X): Y
    def right(b: X, a: Y, c: Y): Y
    def none(b: X, a: Y, c: X): X
```

which we used to define define our Shunt Contraction Scheme:

```
sealed abstract class ShuntContractionScheme[A, B, X, Y]:
    def leaf(v: A): X
    def internal(v: B): Y
    def shuntops: ShuntOperation[X, Y]

    def applyOnTree(tr: Tree[A, B]): X = {
      ...... check details on GitHub repository ......
    }

    def applyOnShuntTree(tr: ShuntTree[A, B]): X = {
      ...... check details on GitHub repository ......
    }

    def applyOnShuntContext(ct: ShuntContext[A, B]): Y = {
      ...... check details on GitHub repository ......
    }
```

However, one disadvantage of tree contraction algorithms is that they completely destroy the input tree. To reconstruct the input tree after the aggregating, Morihata and Matsuzaki (2011) proposed the following:

$$\begin{aligned}
&\mathit{scs}\ \textbf{Tip}\ \mathit{hole}\ \mathit{connect}\ t = t\\
&\textbf{type Context}\ \alpha\,\beta = (\textbf{Tree}\ \alpha\,\beta, \textbf{Tree}\ \alpha\,\beta) \to \textbf{Tree}\ \alpha\,\beta\\
&\mathit{hole}\ :: \forall\alpha,\beta.\beta \to \textbf{Context}\ \alpha\,\beta\\
&\mathit{hole}\ = \lambda(l,r) \to \textbf{Bin}\ l\,a\,r\\
&\mathit{connect}\ :: \forall\alpha,\beta.\ \textbf{Shunt}\ (\textbf{Tree}\ \alpha\,\beta)\,(\textbf{Context}\ \alpha\,\beta)\\
&\mathit{connect}\ = ([\mathit{connect}_L, \mathit{connect}_R, \mathit{connect}_N])\\
&\textbf{where}\ \mathit{connect}_L\ b\ a\ c = \lambda x \to a\,(b\,x, c)\\
&\qquad\quad \mathit{connect}_R\ b\ a\ c = \lambda x \to a\,(b, c\,x)\\
&\qquad\quad \mathit{connect}_N\ b\ a\ c = a\,(b,c)\\
&\qquad\quad x \text{ is of type } (\textbf{Tree}\ \alpha\,\beta, \textbf{Tree}\ \alpha\,\beta)
\end{aligned}$$

Moreover, Morihata and Matsuzaki (2011) formally defines that a function *scs* of the polymorphic type (1) is a Shunt-contraction scheme if

$$\mathit{scs}\ \textbf{Tip}\ \mathit{hole}\ \mathit{connect}\ t = t$$

holds for any tree t. We implemented an equivalent structure that define those functions for tree construction.

From the Shunt-contraction scheme, Morihata and Matsuzaki (2011) proposed the Shunt Tree as a binary tree structure. In our project we also constructed a basic implementation of it, we defined it as follows:

```
// ST a b, ShuntTree
  sealed abstract class ShuntTree[A, B]
  // T a
  case class T[A, B](value: A) extends ShuntTree[A, B]
  // N (ST a b) (ST. a b) (ST a b)
  case class N[A, B](left: ShuntTree[A, B], middle: ShuntContext[A, B], right:
  ↪  ShuntTree[A, B]) extends ShuntTree[A, B]
```

Similarly we defined the related Shunt Context:

```
  // (ST. a b): ShuntContext
  sealed abstract class ShuntContext[A, B]
  // H. b
  case class H[A, B](value: B) extends ShuntContext[A, B]
  // L. (ST. a b) (ST. a b) (ST a b)
  case class L[A, B](left: ShuntContext[A, B], middle: ShuntContext[A, B], right:
  ↪  ShuntTree[A, B]) extends ShuntContext[A, B]
  // R. (ST a b) (ST. a b) (ST. a b)
  case class R[A, B](left: ShuntTree[A, B], middle: ShuntContext[A, B], right:
  ↪  ShuntContext[A, B]) extends ShuntContext[A, B]
```

Their basic functions are size and height which we implemented and proved for both Tree and Context.

ST corresponds to **Tree**, while **ST$_\bullet$** corresponds to **Context**. One can transform a tree t to its Shunt-Tree representation by *scs* **T H$_\bullet$** ([**L$_\bullet$ R$_\bullet$ N**]) t, while reconstructing the tree structure via $s2t$ and $s2c$, which we implemented as follows:

```
def s2t[A, B](st: ShuntTree[A, B]): Tree[A, B] = {
    st match {
      case T(v) => Tip(v)
      case N(b, a, c) => {
        val lt = s2t(b)
        val rt = s2t(c)
        connectN(lt, s2c(a), rt)
      }
    }
  }

  def s2c[A, B](sc: ShuntContext[A, B]): Context[A, B] = {
    sc match {
      case H(v) => hole(v)
      case L(b, a, c) => connectL(s2c(b), s2c(a), s2t(c))
      case R(b, a, c) => connectR(s2t(b), s2c(a), s2c(c))
    }
  }
```

3.5 Conclusion and Future Work

3.5.1 Rundown of Our Work

In this project, we addressed the limitations of existing data structures in functional programming for parallel computation by developing practical implementations inspired by the theoretical foundations proposed by Morihata and Matsuzaki (2011). Specifically, we:

- **Developed the basis for Join Lists**: This included both unbalanced and balanced implementations, where we implemented recursive functions to facilitate parallel computation while proving their correctness.
- **Explored Binary Trees**: We introduced a polymorphic tree data structure, implementing key operations such as mapping, aggregation, and type conversions while ensuring correctness through formal proofs.
- **Implemented Shunt Trees**: Leveraging the Shunt contraction scheme, we created data structures optimized for parallel processing, along with associated functions for tree transformation and contraction.
- **Provided Verified Implementations**: All implemented methods were formally verified using Stainless, ensuring their adherence to theoretical guarantees of correctness.
- **Addressed Parallelization Challenges**: We demonstrated how balanced data structures like Join Lists and Shunt Trees enable efficient divide-and-conquer parallelism by reducing the depth of recursive operations to logarithmic bounds.

Almost all the theorems and properties we discussed in this document were formally implemented and verified.

3.5.2 Improvements and Extensions

While our work lays a strong foundation for parallel computation using functional programming constructs, there are several opportunities for expansion and refinement:

1. **Generalizing the Shunt Tree Framework**:
   - Extend the Shunt Tree data structure to support arbitrary tree shapes and operations beyond the current limited binary tree context. This includes supporting insertions, deletions, and rebalancing efficiently.
2. **Extending Aggregation Functions**:
   - Explore additional applications of the Shunt contraction scheme. For example, implementing operations such as `zip`, advanced reductions, and mapping functions tailored to Shunt Trees.
3. **Scaling to System-Level Parallelism**:
   - Implement parallel computation benchmarks to empirically validate the theoretical speedup of our proposed data structures on multi-core processors.

References

[1] Bird, R. S. (1987). An introduction to the theory of lists. Logic of Programming and Calculi of Discrete Design, 5–42. `https://doi.org/10.1007/978-3-642-87374-4_1`

[2] Morihata, A., & Matsuzaki, K. (2010). Automatic parallelization of recursive functions using quantifier elimination. Proceedings of the 10th International Conference on Functional and Logic Programming (FLOPS'10), 321–336. `https://doi.org/10.1007/978-3-642-12251-4_23`

[3] Morihata, A., & Matsuzaki, K. (2011). Balanced trees inhabiting functional parallel programming. Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, 117–128. `https://doi.org/10.1145/2034773.2034791`

[4] Reif, J. H. (1993). Synthesis of parallel algorithms. Morgan Kaufmann Publishers Inc.

A List of Implemented Methods for JoinList

Function	Description
`==(other: JoinList[T]): Boolean`	Check if this and other list are equal
`toList: List[T]`	Turn a JoinList to a stainless List
`size: BigInt`	Number of elements in this JoinList
`apply(i: BigInt): T`	Return the element in index i in this JoinList
`content: Set[T]`	The set of elements in this JoinList
`contains(x: T): Boolean`	Returns true if this JoinList contains x
`isEmpty: Boolean`	Check if the JoinList is empty
`head: T`	Returns the head of this (non-empty) JoinList.
`tail: JoinList[T]`	Returns the tail of this (non-empty) List
`::(t: T): JoinList[T]`	Prepend an element to this JoinList
`:+(t: T): JoinList[T]`	Append an element to this JoinList
`take(i: BigInt): JoinList[T]`	Take the first i elements of this JoinList, or the whole JoinList if it has less than i elements
`drop(i: BigInt): JoinList[T]`	This List without the first i elements, or the *Empty()* if this List has less than i elements.
`slice(from: BigInt, to: BigInt): JoinList[T]`	Take a sublist of this JoinList, from index from to index to
`foldLeft(z: R)(f: (R, T) => R): R`	Applies the binary operator f to a start value z and all elements of this JoinList, going left to right
`foldRight(z: R)(f: (T, R) => R): R`	Applies a binary operator f to all elements of this JoinList and a start value z, going right to left.
`map(f: T => R): JoinList[R]`	Builds a new JoinList by applying a predicate f to all elements of this JoinList.
`++(other: JoinList[T]): JoinList[T]`	Append this JoinList with *other*
`-(that: JoinList[T]): JoinList[T]`	Remove all occurrences of any element in *that* from this JoinList
`&(that: JoinList[T]): JoinList[T]`	A JoinList of all elements that occur both in *that* and this List

Table 1: Function Names and Signatures