Efficient Hash-Consing of Recursive Types

Jeffrey Considine
jconsidi@cs.bu.edu

January 29, 2000

Abstract

Efficient storage of types within a compiler is necessary to avoid large blowups in space during compilation. Recursive types in particular are important to consider, as naive representations of recursive types may be arbitrarily larger than necessary through unfolding. Hash-consing has been used to efficiently store non-recursive types [7]. Deterministic finite automata techniques have been used to efficiently perform various operations on recursive types [4]. We present a new system for storing recursive types combining hash-consing and deterministic finite automata techniques. The space requirements are linear in the number of distinct types. Both update and lookup operations take polynomial time and linear space and type equality can be checked in constant time once both types are in the system.

1 Introduction

A recent trend in compilers is the use of typed intermediate languages while compiling to generate safer and more optimized code. One disadvantage to this approach is that saving type information can easily cause a large blowup in the space used by a compiler. Types of simple expressions can blowup exponentially when represented using a tree data structure. Recursive types allow arbitrary expansion through simple unfolding.

We use the idea of hash-consing to maintain a set of types in which all equivalent types share the same representation. [7] reports sucessful applications of these techniques. We take the idea one step farther and add DFA techniques to allow the same benefits to be extended to recursive types.

DFA techniques have been used for various operations on recursive types [4]. We use them to minimize recursive type representations and avoid duplicate representations. We also present a canonical ordering algorithm to avoid the "isomorphic under permutations of states" clause in many presentations of DFA minimization algorithms. Use of canonical orderings allows linear time DFA comparisons and simplifies comparing recursive types.

Given a set S of types and a type τ, our system allows efficient update and lookup operations to be performed in two steps: a pre-processing step taking $O(|\tau|^2)$ time and a step actually accessing S using $O(|\tau| \log |S|)$ time. The space required to represent S is linear in the number of reachable distinct types so it is asymptotically optimal in space.

This work was done during the period of October 1999 to December 1999. In the mean time, Laurent Mauborgne defended his PhD thesis on representing sets of infinite trees [5] and submitted an excerpt [6] for publication in ESOP 2000. The core algorithm for efficiently constructing infinite trees is virtually identical to the algorithm given here for representing recursive types, though the presentation is drastically different. The key difference between the two representations is the space utilization - ignoring logarithmic factors, the size used per distinct type is constant in the representation presented here and dependent on the number of mutually recursive trees in the representation of Mauborgne.

2 Recursive Types

2.1 Syntax

The common syntax of *recursive types* is

$$\tau ::= b \mid \tau \to \tau \mid \mu v.\tau \mid v$$

where b is a base type and v is a type variable. We generalize this syntax to include arbitrary type constructors, in addition to "$\to$".

Let B be a finite set of base types and C a finite set of type constructors. B and C are disjoint. Let V be the set of type variables.

$$\tau ::= b \mid c(\tau, \ldots) \mid \mu v.\tau \mid v$$

where $b \in B$, $c \in C$, and $v \in V$.

2.2 Equivalence

Informally, we define two types to be equivalent if their corresponding labeled infinite trees are the same. We will formally define the labeled infinite tree corresponding to a recursive type in 3.3.

3 Infinite Trees

In describing infinite trees, we will use the notation that $\vec{i}$, $\vec{j}$, and $\vec{k}$ are strings of positive integers and i, j, and k are individual positive integers. Let P be the set of positive integers. ϵ is the empty string and P^* is the set of all strings of positive integers. $\cdot$ is an overloaded string concatenation operator. Given $I, J \in P^*$, $I \cdot J = \left\{\vec{i} \cdot \vec{j} | (\vec{i} \in I) \wedge (\vec{j} \in J)\right\}$.

3.1 Trees

Definition 3.1 (Trees) T is a *tree* if the following are true:

1. T is a non-empty set of strings built from positive integers - $\emptyset \subset T \subseteq P^*$.
2. T is prefix-closed - $\forall \vec{i}, \vec{j} \left(\left(\vec{i} \cdot \vec{j} \in T\right) \to \left(\vec{i} \in T\right)\right)$.
3. T is leftward-closed - $\forall \vec{i}, j, k \left((j \leq k) \to \left(\vec{i} \cdot k \in T\right) \to \left(\vec{i} \cdot j \in T\right)\right)$.

We call the strings in T the *paths* of T. Note that the first two properties imply that ϵ is a path of T. In words, ϵ identifies the root node of T and T is a potentially infinite set of finite paths.

Definition 3.2 (Subtrees) Let T be a tree and $\vec{t}$ a path in T. The *subtree* of T reached by $\vec{t}$ is defined as follows:

$$\textit{subtree}\left(T, \vec{t}\right) = \left\{\vec{i} \mid \left(\vec{t} \cdot \vec{i} \in T\right)\right\}$$

Lemma 3.3 *Let T be a tree and $\vec{t}$ a path of T. Then, $\textit{subtree}(T, \vec{t})$ is a tree.*

Proof Sketch:

If $\vec{s}$ is a witness that $\textit{subtree}(T, \vec{t})$ is not a tree, $\vec{t} \cdot \vec{s}$ is similarly a witness that T is not a tree.

Definition 3.4 (Degree) Let T be a tree. The *degree* of T is defined as follows:

$$\textit{degree}(T) = |T \cap P|$$

In words, $\textit{degree}(T)$ is the number of children of the root node of T.

3.2 Labeled Trees

Definition 3.5 (Labeled Trees) Let *arity* be a function from C to P returning the arity of its input and let k be the maximum value returned by *arity*. We define *labeled trees* as follows - $\mathcal{T} = (T, l)$ is a tree if the following are true:

1. T is a tree.
2. l is a function from P^* to $B \cup C$ such that
   - $\forall t((t \in T) \to (l(t) \in B) \to (degree(subtree(T,t)) = 0)$
   - $\forall t((t \in T) \to (l(t) \in C) \to (degree(subtree(T,t)) = arity(l(t)))$

Definition 3.6 (Labeled Subtrees) Let $\mathcal{T} = (T, l)$ be a labeled tree and $\vec{t}$ a path of T. The *labeled subtree* of $\mathcal{T}$ reached by $\vec{t}$ is defined as follows:

$$labeledsubtree\left(T, \vec{t}\right) = \left(\left\{\vec{i} \mid \left(\vec{t} \cdot \vec{i} \in T\right)\right\}, \lambda x.l\left(\vec{t}x\right)\right)$$

Lemma 3.7 *Let $\mathcal{T} = (T, l)$ be a labeled tree and $\vec{t}$ a path of T. Then, $labeledsubtree\left(\mathcal{T}, \vec{t}\right)$ is a labeled tree.*

Definition 3.8 (Labeled Paths) Let $\mathcal{T} = (T, l)$ be a labeled tree. The *labeled paths* of $\mathcal{T}$ is defined as follows:

$$paths(\mathcal{T}) = \left\{\left(\vec{t}, x\right) \mid \left(\vec{t} \in T\right) \wedge (x = l(t))\right\}$$

Definition 3.9 (Labeled Tree Isomorphism) Let $\mathcal{T}_1 = (T_1, l_1)$ and $\mathcal{T}_2 = (T_2, l_2)$ be labeled trees. $\mathcal{T}_1$ and $\mathcal{T}_2$ are isomorphic iff $T_1 = T_2$ and $\forall \vec{t}\left(\left(\vec{t} \in T_1\right) \to \left(l_1\left(\vec{t}\right) = l_2\left(\vec{t}\right)\right)\right)$.

3.3 Labeled Tree Construction from Recursive Types

Definition 3.10 (unfold) Let τ be a recursive type. We define the function *unfold* as follows:[1]

$$unfold(\tau) = \begin{cases} [\mu t.\sigma/t]\sigma \text{if } \tau = \mu t.\sigma \\ \tau \text{otherwise} \end{cases}$$

Definition 3.11 (rec_unfold) Let τ be a recursive type. We define the function *rec_unfold* as follows:[2]

$$rec\_unfold(\tau) = \begin{cases} b & \text{if } \tau = b \in B \\ c(rec\_unfold(\tau_1), \ldots) & \text{if } \tau = c(\tau_1, \ldots) \\ v & \text{if } \tau = v \in V \\ unfold(\mu t.rec\_unfold(\sigma)) & \text{if } \tau = \mu t.\sigma \end{cases}$$

Definition 3.12 (Finite Trees of Recursive Types) Let τ be a recursive type. We define the finite tree of τ as follows:

$$finite\_tree(\tau) = \begin{cases} \{\epsilon\} & \text{if } \tau \in B \\ \cup_i (i \cdot finite\_tree(\tau_i)) \cup \{\epsilon\} & \text{if } \tau = c(\tau_1, \ldots) \\ \{\epsilon\} & \text{if } \tau \in V \\ finite\_tree(\sigma) & \text{if } \tau = \mu t.\sigma \end{cases}$$

In words, *finite_tree*(τ) is the finite set of all finite paths traversing τ without unfolding. Later, a similar but potentially infinite set will be described.

[1] The common axiom $\mu t.\sigma \cong [\mu t.\sigma/t]\sigma$, where $\cong$ denotes equivalence, allows one to prove $\tau \cong unfold(\tau)$.

[2] As with *unfold*, $\tau \cong rec\_unfold$.

Lemma 3.13 *Let τ be a recursive type. Then, $\mathit{finite\_tree}(\tau)$ is a tree.*

Proof Sketch:
This lemma may be proven using structural induction over τ.

Lemma 3.14 *Let τ be a recursive type. Then, $\mathit{finite\_tree}(\tau) \subseteq \mathit{finite\_tree}(\mathit{rec\_unfold}(\tau))$.*

Definition 3.15 (Infinite Trees of Recursive Types) Let τ be a recursive type. Let $\tau_i = \mathit{rec\_unfold}^i(\tau)$, the result of applying *rec_unfold* to τ i times. Let $p_i = \mathit{finite\_tree}(\tau_i)$. The infinite tree of τ is defined as follows:

$$\mathit{infinite\_tree}(\tau) = \cup_i p_i$$

In words, $\mathit{infinite\_tree}(\tau)$ is the potentially infinite set of all finite paths traversing τ unfolded an infinite number of types. If τ is not recursive, $\mathit{infinite\_tree}(\tau) = \mathit{finite\_tree}(\tau)$, a finite set.

Lemma 3.16 *Let τ be a recursive type. Then, $\mathit{infinite\_tree}(\tau)$ is a tree.*

Proof Sketch:
This lemma follows from lemmas 3.13 and 3.14.

Lemma 3.17 *Let τ be a recursive type. Then, $\mathit{infinite\_tree}(\tau)$ is decidable.*

Proof Sketch:
Note that $\vec{t} \in \mathit{infinite\_tree}(\tau)$ iff $\vec{t} \in \mathit{finite\_tree}(\mathit{rec\_unfold}^{|\vec{t}|}(\tau))$.
Let Ω be a distinguished symbol which stands for an undefined result.

Definition 3.18 (Finite Labeling of Recursive Types) Let τ be a recursive type and x a path. We define the finite labeling of τ as follows:

$$\mathit{finite\_labeling}(\tau)(x) = \begin{cases} b & \text{if } \tau = b \in B \text{ and } x = \epsilon \\ \mathit{finite\_labeling}(\tau_i)(x') & \text{if } \tau = c(\tau_1, \ldots) \text{ and } x = i \cdot x' \\ \Omega & \text{if } \tau \in V \\ \mathit{finite\_labeling}(\sigma)(x) & \text{if } \tau = \mu t.\sigma \end{cases}$$

Definition 3.19 (Infinite Labeling of Recursive Types) Let τ be a recursive type and x be a path. The infinite labeling of τ is defined as follows:

$$\mathit{infinite\_labeling}(\tau)(x) = \begin{cases} y & \text{if there exists } i \text{ such that } \mathit{finite\_labeling}(\mathit{rec\_unfold}^i(\tau))(x) = y \\ \Omega & \text{otherwise} \end{cases}$$

Note $\mathit{infinite\_labeling}(\tau)$ has an infinite domain iff $\mathit{infinite\_tree}(\tau)$ is infinite.

Lemma 3.20 *Let τ be a recursive type. Then, $\mathit{infinite\_labeling}(\tau)$ is a well defined and total computable function.*

Proof Sketch:
Note $\mathit{finite\_labeling}(\tau)(x) = y$ implies $\mathit{finite\_labeling}(\mathit{rec\_unfold}(\tau))(x) = y$ and the number of applications of *rec_unfold* necessary is bounded by the length of x.

Lemma 3.21 *Let τ be a recursive type. Let $T = \mathit{infinite\_tree}(\tau)$ and $l = \mathit{infinite\_labeling}(\tau)$. Then, $\mathcal{T} = (T, l)$ is a labeled tree.*

Definition 3.22 (Recursive Type Labeled Infinite Trees) Let τ be a recursive type. Let $T = \mathit{infinite\_tree}(\tau)$ and $l = \mathit{infinite\_labeling}(\tau)$. The labeled infinite tree corresponding to τ is defined to be $\mathcal{T} = (T, l)$.

4 Deterministic Finite Automata

Other work has been done using *deterministic finite automata* (DFA's) to efficiently implement operations such as subtyping [4]. We start with DFA's for processing strings and modify them to establish a correspondence with types in our system. We can then use adaptations of standard DFA algorithms to minimize the size of the types and traditional graph algorithms to analyze relations between types.

4.1 Standard DFA's

Traditionally, a string processing DFA can be considered as a directed multi-graph with each edge labeled with a member of the input alphabet. Each node corresponds to a state of the DFA and is labeled final or non-final. Strictly speaking, each node has exactly one outgoing edge for each member of the input alphabet. However, it is common practive to leave out edges to "error" states, non-final states in which all edges are self-loops, since once an error state is reached, it is impossible to reach a final state [3].

When processing strings with standard DFA's, one node of the graph is designated as the start state of the DFA. Input to the DFA is read one character at a time and the corresponding edge is followed. When the end of input is reached, the sDFA accepts the input string iff last state of the traversal is a final state.

We avoid a formal definition of DFA's which can be found in many textbooks on automata theory. Unless otherwise specified, we follow the terminology of [3].

In contrast to the modified DFA's of 4.2, we will refer to standard DFA's as sDFA's.

4.2 Modified DFA's

Instead of using sDFA's, we use a *modified DFA* (mDFA) with a more general labeling scheme beyond final/non-final. mDFA's accept strings built from members of $\{1, \ldots, k\}$. Each state is labeled with the identifier of either a type constructor or a base type. In 4.3, it will be seen that type constructor labels and base type labels will be distinguishable by the number of outgoing edges. The presence of a sink state is implied but not used; all missing edges go to this sink state.

String processing with mDFA's is similar to that with sDFA's. Traversal of the mDFA graph is the same as with a DFA, but the output is different. If the last state reached is a sink state, the mDFA rejects. Otherwise, it accepts and outputs the state's label.

Definition 4.1 (mDFA Output) Let A be an mDFA. The *output* of A is the set of pairs of strings accepted by A and the associated output.

$$\mathit{output}(A) = \{(\vec{x}, y) \mid A \text{ accepts } \vec{x} \text{ and } A \text{ outputs } y \text{ given } \vec{x}\}$$

4.3 mDFA Construction from Recursive Types

Definition 4.2 (mDFA/Recursive Type Equivalence) Let τ be a recursive type and A be an mDFA. Let $\mathcal{T}$ be the labeled tree corresponding to type τ. We define A and τ to be equivalent iff $\mathit{output}(A) = \mathit{paths}(\mathcal{T})$.

To convert a recursive type τ into an mDFA A, the tree structure of τ is essentially copied into an mDFA graph and back references are added to replace bound type variables.

Lemma 4.3 (mDFA Construction from Recursive Types Algorithm) *Let τ be a recursive type. There exists an algorithm running in $O(|\tau| \log |\tau|)$ time and $O(|\tau|)$ space outputting mDFA A such that A is equivalent to τ and if A has n states, $n \in O(|\tau|)$.*

Proof Sketch:

A can be built using two traversals of the tree representation of τ. The first pass assigns states to components of τ and the second builds A.

The first pass annotates the tree representation of τ with state numbers in a bottom up fashion. Fresh states are assigned to base types and applications of type constructors and passed back up the tree. Type bindings are annotated with the state passed up by the body of binding and this state is passed up the tree again. This pass takes $O(|\tau|)$ time. Bound type variables are not assigned states.

The second pass actually generates the adjacency list representation of the mDFA graph. τ is traversed in a depth first manner passing a binding environment down to each child. When a base type is reached, an appropriately labeled state with no outgoing edges is created. When an application of a type constructor is reached, an appropriately labeled state with a numbered edge to each child is created; if the child was not assigned a state in the first pass, it is a bound type variable and looked up in the binding environment (if it is not in the binding environment, τ is invalid). When a type binding is reached, a binding of the type variable to the assigned state is added to the environment that will be passed to the body of the type binding. Again, no action is taken when a bound type variable is reached. This pass takes $O(|\tau| \log |\tau|)$ time using a balanced tree representation for the environment.

This process preserves the tree structure of τ except when type variables are bound and used. Ignoring uses of bound type variables for a moment, the semantics of a type binding give it the same structure as its body so sharing the state of its body is semantically correct. Considering uses of bound type variables again, the semantics of using a bound type variable are that it is equivalent to the inner most binding of that type variable so using the state the type variable is currently bound too is also correct.

4.4 Minimization of mDFA's

Definition 4.4 (mDFA Equivalence) Let A and B be mDFA's. Then, A and B are defined to be equivalent iff $output(A) = output(B)$.

Lemma 4.5 (mDFA Minimization Algorithm) *Let A be an mDFA with n states. There exists an algorithm running in $O(n \log n)$ and $O(n)$ space outputing an mDFA B such that A and B are equivalent and B is minimal. That is, for all mDFAs C, if A and C are equivalent and C has m states, then $n \leq m$.*

Proof Sketch:

An sDFA is usually minimized by calculating the equivalence relation or equivalence classes of states [8].[3] Once the equivalence classes of an sDFA A are known, it is straightforward to create a minimal equivalent sDFA B using a homomorphism mapping each equivalence class of A to a distinct state in B.

The first approximation of the equivalence classes is a partitionning of the states according to whether they are labeled final or not. This approximation is refined further by splitting equivalence classes according to the states various inputs bring them to. If there are n states in $|A|$, at most n splits are necessary, and there are many sDFA minimization algorithms running in $O(n^2)$ time. Most of these sDFA minimization algorithms also run in $O(n)$ space.[4]

Adapting these algorithms to mDFA's simply consists of replacing the original partitionning with one according to the mDFA labels. There may be as many sets in the partition as there are type constructors and base types, as opposed to merely two with sDFA's. This partitionning is exactly that produced by checking 0-distinguishability.[5] Since this is the criteria used for proving the correctness of the original partitionning in sDFA algorithms, the correctness proof for the sDFA algorithm can be trivially transformed into a correctness proof for the mDFA algorithm. This gives many algorithms for transforming an mDFA A into a minimal equivalent mDFA B in $O(n^2)$ time and $O(n)$ space. Hopcroft's $O(n \log n)$ time algorithm can be transformed in the same fashion.

[3] A notable exception is the algorithm in [3] which calculates the difference relation from which it is trivial to calculate the equivalence relation.

[4] The algorithm in [3] is an exception again since it uses $O(n^2)$ space to calculate the distinguishability relation.

[5] A and B are k-distinguishable if there is a string of length k that is a counter example to A being equivalent to B.

4.5 Reconstruction of Recursive Types from mDFA's

Once τ has been converted into an mDFA A and minimized to get another mDFA B, we may want to reconstruct a minimal tree representation of a type σ which is equivalent to τ.

Lemma 4.6 (Reconstruction of Recursive Types from mDFA's) *Let A be an mDFA with n states. There exists an algorithm outputing a recursive type τ such that A and τ are equivalent, τ is minimal, and the algorithm runs in $O(|\tau| \log n)$ and $O(|\tau|)$ space.*

Proof Sketch:
The tree structure of σ can be generated by a depth first traversal of B allowing nodes to be visited more than once. An environment containing all ancestors in the traversal is passed down the traversal to detect back references. Subtrees and the set of all states referenced is passed back up the tree.

At each state visited, if the state is an ancestor of itself, the state is converted into a bound variable and passed back up the tree with the singleton set of that state being the only referenced state. If the state is not an ancestor of itself and has a base type label, the base type is passed back with an empty set of referenced states. If the state is not an ancestor of itself and has a type constructor label, its children are generated and their sets of free references are unioned together. The type constructor is applied to the child subtrees. If the state is in the set of free references, a type binding is added and it is removed from the set of free references. Both the type and set of free references are then passed back.

All set operations except union take $O(\log |B|)$ time. Union takes $O(|B|)$ time, so $O(|B|)$ is spent at each node of σ. The entire process takes $O(|\sigma||B|)$ time. In terms of τ, this is $O(|\tau|^2)$ time.

σ will have the same infinite tree structure as τ, but each infinite branch will be terminated at the first node equivalent to a parent. The tree structure of σ is provably minimal type since earlier termination implies different structure from τ and later termination is not minimal. The μ bindings in σ are determined by the nodes at which σ is terminated: the μ bindings present are exactly the bindings used.

4.6 Graph Properties

It is interesting to consider some properties of the graph induced by the mDFA edges. We mention some without proof but they should be intuitive.

- If a state can reach itself, the corresponding type is recursive.
- If two states can reach each other, their corresponding types are mutually recursive.
- If one state can not reach a second state, then the type corresponding to the first state may be analyzed independently of the type corresponding to the second.

5 Hash-Consing

Historically, hash consing is a technique originally used in LISP to avoid duplication of lists. In LISP, list structures are only created by the cons operation. By modifying cons with the help of hashing techniques, no two invocations of cons would ever return distinct copies of the same data. An early example of this technique is presented in [2]. While limiting the ability to modify lists generated in such a manner, this technique allows greater space efficiency and constant time equality checking [1].

We use similar ideas to represent a set of types S. The equivalence class $c(\tau)$ of a type τ is the set of all types equivalent to τ; a set C of types is an equivalence class if $C = c(\tau)$ for some type τ. We think of S as a finite collection of equivalence classes, where each equivalence class is uniquely represented by a canonical member of the class and accessed through a unique handle. Internally, S is represented by

- A mapping from base type identifiers to handles (natural numbers). Given a total order of base types, this mapping accesible in $O(S)$ time. Typically there will be a fixed number of base types so this will be accessible in $O(1)$ time.

- A mapping from tuples of type constructors and the handles of their children to handles. Given a total order of type constructors, there is a total order of these tuples and this mapping is accessible in $O(\log |S|)$ time.
- A mapping from canonically ordered mDFA's to handles. These mDFA's have a slight modification from the mDFA's discussed in 4.2: base type identifiers in labels are replaced with hash-cons handles. Since the states of the mDFA's have a canonical order, the mDFA's can be given a meaningful total order. For an mDFA A, this mapping is accessible in $O(|A| \log |S|)$ time.

  Semantically, each mDFA maps to the handle of its first state and successive states are mapped to successive handles.
- A mapping inverting the union of the previous mappings. This mapping can be sorted by handle and accessed in $O(\log |S|)$ time.

5.1 Hash-Consed Recursive Types

When hash-consing recursive types, two modifications from the syntax given in 2.1 are used. First, handles of hash-consed types are added to the allowed syntax for recursive types. Second, free type variables are replaced with extra base types to allow uniform treatment of type variables in 5.4.

5.2 Hash-Consing Base Types

Lemma 5.1 *There is an algorithm to hash-cons base types in $O(\log |S|)$ time. If the set of base types is finite and fixed, there is an algorithm to hash-cons base types in $O(1)$ time.*

Proof Sketch:
Base types are a trivial case. Since base types are unrelated to other types, the data structures of S can be updated in $O(\log |S|)$ time. If the set of base types is fixed and the base types are hash-consed ahead of time, this improves to $O(1)$ time.

5.3 Hash-Consing Type Constructor Applications

Lemma 5.2 *Let τ be a recursive type without any type bindings or bound type variables. There is an algorithm to hash-cons τ in $O(|\tau| \log |S|)$ time.*

Proof Sketch:
τ can be hash-consed in a bottom up manner using $O(\log |S|)$ time per subtree. The total time is $O(|\tau| \log |S|)$.

5.4 Hash-Consing Recursive Types

Lemma 5.3 *Let τ be a recursive type. Let h be the last hash-cons handle[6] mentionned in τ. Let m be the number of states of the mDFA refered to by h or 0 if h is not recursive or there are no hash-cons handles in τ. There is an algorithm to hash-cons τ in $O(|\tau|^2 + |\tau|m + |\tau| \log |S|)$ time.*

Proof Sketch:
It is useful to separate the subtrees of τ which have no free variables and consider them independently. When calculating sets of free variables in the traditional bottom up fashion, any subtree of τ with no free variables can be considered independently of the rest of τ. These subtrees can be hash-consed and replaced with their handle leaving a smaller but equivalent version of τ to examine.

[6]Hash-cons handles are ordered in the same way as natural numbers and allocated squentially in increasing order.

Suppose σ is such a subtree of τ. If σ's children have no free variables, then σ can be hash-consed normally as discussed in 5.2 and 5.3. Otherwise, σ is a recursive type and the subtrees of σ that have not been hash-consed represent a set of mutually recursive types. This is easily verified by induction over the structure of σ by noting that each such subtree has at least one free variable which must refer to an ancestor in the tree.

σ is converted into an mDFA[7] A and minimized with a canonical ordering to get mDFA B. All states in B not labeled with handles correspond to types that are mutually recursive with each other. However, it is not known yet whether states remaining to be hash-consed are mutually recursive with any previously hash-consed states.[8]

Checking all states of hash-consed mDFA's represented by handles in σ is prohibitively expensive: there are $O(|B|)$ possible handles, $O(|B|)$ states that need to be checked for each possible form of mutual recursion, and the size m of the largest mDFA represented by a handle is arbitrarily large. In practice, these checks should be prunable but this is still $O(|B|^2 m)$ time. By noting that only the mDFA represented by the greatest handle in σ can be mutually recursive with the states of B,[9] this can be brought down to $O(|B|m)$ time.

If σ is mutually recursive with a previously hash-consed state, then the certificate (a successful traversal mapping σ to previously hash-cons handles) gives the handle of σ. Otherwise hash-consing σ and B's states remaining to be hash-consed in $O(|\sigma| \log |S|)$ time.

Once σ is separated from τ, $O(|\sigma|m + |\sigma| \log |S|)$ time is spent hash-consing σ. Separating τ into subtrees takes $O(|\tau|^2)$ time and the total time hash-consing τ is $O(|\tau|^2 + |\tau|m + |\tau| \log |S|)$. When τ has no hash-cons handles, this is $O(|\tau|^2 + |\tau| \log |S|)$ and $O(|\tau|)$ space.

Theorem 5.4 (Recursive Type Hash-Consing Algorithm) *Let τ be a recursive type with no references to hash-consed types. There is an algorithm to hash-cons τ in $O(|\tau|^2 + |\tau| \log |S|)$ time and $O(|\tau|)$ space.*

Proof Sketch:

This theorem follows from lemmas 5.1, 5.1, and 5.3.

5.5 Hashing

Hashing techniques have been ignored so far but can be used to speed up the average case of various lookups. [7] reports great success with using hashing in their hash-consing scheme. If the base types and type constructors are fixed before hand, the mappings from hash-cons entry to handle can be split according to base type or type constructor and the correct mapping can be chosen in constant time.

6 Conclusion

We have presented an efficient system for managing types. Key features of this system include asymptotically optimal space usage and constant time type equality tests. While the component ideas of hash-consing and representing recursive types as DFA's have both been used before, we believe that their combination to support recursive types while hash-consing is new.

An earlier form of this system has been implemented in SML. All changes since this implementation have been both simplifications and algorithmic improvements. Later, we plan to adapt this system for use in the Church Project SML compiler.

[7] σ is converted into a variant of mDFA's using hash-cons handles in place of base type identifiers

[8] A recursive type may be defined in terms of a previous definition of itself.

[9] The manner in which handles are assigned to mDFA's embeds the partial order induced by mDFA dependencies.

7 Acknowledgements

This report was written to summarize some work done in a Directed Study with my advisor, Assaf Kfoury. The original motivation for work on this problem was given by Joe Wells. Allyn Dimock was helpful in explaining the context of the Church compiler into which this work will be integrated.

A DFA Canonical Ordering

Definition A.1 (DFA Canonical Ordering) A function $f : DFA \rightarrow DFA$ gives a canonical ordering to mDFA's iff forall mDFAs A and B, A and B are equivalent iff $f(A)$ and $f(B)$ are identical.[10] This definition applies to all DFA variants we have discussed.

Lemma A.2 (DFA Canonical Ordering Algorithm) *Given an mDFA A with n states, there is an algorithm running in $O(n^2)$ time and $O(n)$ space outputing B such that B is minimal and $B = f(A)$ for some canonical ordering f.*

Proof Sketch:

We present an algorithm to minimize a DFA to produce an minimum isomorphic DFA with a canonical state ordering. The algorithm given is for standard DFA's, but is applicable to all DFA variants we have mentionned. The algorithm constructs a canonically ordered DFA in a manner similar to the way it is minimized - the main distinction from DFA minimization is that the partitions approximating the equivalence relation are ordered internally.

First, the sets of the original partition are totally ordered, placing the set of final sets before the set of non-final sets. As sets are split according to the previous partitionning, the subsets are ordered according to the sets that split them. Details of this splitting and ordering process follow.

Approximations of the equivalence relation are represented as lists of sets of states. Given such a list L, the n states are given a partial ordering by mapping all states in the ith set in L to i. States are compared by comparing their associated values. The mapping can be created in $O(n)$ time and accessed in $O(1)$ time using an array indexed by state number.

An element S of L is split if the members of S do not have their children mapped to the same sets. That is, each element s of S is characterized by the values its children are mapped to. The characterization of an element s can be determined in $O(1)$ time since there is a constant upper bound on the number of children. Checking whether S needs to be split (i.e. checking whether some of the characterizations are different) can be done in $O(|S|)$ time. If S needs to be split, it can be split in $O(n)$ time using pigeon-hole sort. When s is split, the subsets are ordered according to the children that split them. Splits based on different children are done separately in a consistent order.

The total number of approximations is at most n since there are at most n non-isomorphic states and each approximation must add at least one split. The total time checking an approximation for splits is $O(n)$ time so the total time checking for splits is $O(n^2)$. The total time performing splits is $O(n^2)$ since there are at most n splits and each split takes $O(n)$ time. Therefore, the total time is $O(n^2)$.

The proof of this algorithm's correctness is similar to that of the corresponding minimization which does not order the sets. The kth approximation separates states that are k-distinguishable. It is trivial to verify that the states are ordered by the first string distinguishing them (note states distinguished by shorter strings are already ordered).[11]

References

[1] John Allen. Anatomy of LISP. McGraw-Hill Book Company, New York, 1978.

[10] Here, we say two DFAs are identical if their internal representations including labeling are identical.

[11] Here, we order strings primarily by length and secondarily by lexicographic order.

[2] E. Goto. Monocopy and Associative Algorithms in an Extended Lisp. University of Tokyo, Japan, May 1974.

[3] John E. Hopcroft, Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company, Inc., Reading, Massachuesetts, 1979.

[4] Dexter Kozen, Jens Palsberg, Michael I. Schwartzbach. Efficient Recursive Subtyping. In Mathematical Structures in Computer Science, 5(1):113-125, 1995.

[5] Laurent Mauborgne. Representation of Sets of Trees for Abstract Interpretation. http://www.di.ens.fr/ mauborgn/t.ps.gz, November 1999.

[6] Laurent Mauborgne. Improving the Representation of Infinite Trees to Deal with Sets of Trees. to appear in ESOP 2000.

[7] Zhong Shao, Christopher League, and Stefan Monnier. Implementing Typed Intermediate Languages. In Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98), Baltimore, Maryland, pages 313-323, September 1998.

[8] Bruce W. Watson. A taxonomy of finite automata minimization algorithms. Computing Science Report 93/44, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1993.