The Wizard of TILT: Efficient$^?$, Convenient, and Abstract Type Representations

Tom Murphy

March 2002

CMU-CS-02-120

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Senior Thesis Advisors
Bob Harper (rwh@cs.cmu.edu)
Karl Crary (crary@cs.cmu.edu)

Abstract

The TILT compiler for Standard ML is type-directed and type-preserving, that is, it makes use of and translates type information during the phases of compilation. Unfortunately, such use of type data incurs a significant overhead. This paper explores methods for abstractly, conveniently, and efficiently storing and manipulating type information in TILT. In the end, we discover that doing *more* work to reduce overhead is a bad strategy for this situation.

This material is based on work supported in part by ARPA grant F-19628-95-C-0050 and NSF grant CCR-9984812. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of these agencies.

Keywords: types, typed intermediate languages, type representations, Standard ML, hash consing, de Bruijn indices, views

1 The Wizard of TILT

TILT is a certifying compiler for Standard ML [1]. Its major distinguishing feature is the use of Typed Intermediate Languages throughout the phases of compilation. Because each of the code transformations that the compiler performs also transforms the types, we preserve type information that is normally discarded after typechecking the source language in traditional compilers. This allows us to typecheck the results of these transformations (catching compiler bugs), perform data representation optimizations, and do nearly tag-free garbage collection. We eventually intend for TILT to generate proof-carrying code [2].

Unfortunately, storing and processing types at compile-time imposes a performance penalty on the compiler. With type-checking enabled after each transformation and optimization, TILT is slow.

This paper recounts our experience in attempting to implement a more efficient type representation strategy into the substantial existing code base. Though the abstraction and optimizations are successful, in the end we are overwhelmed by the overhead necessary to implement them.

2 The Curtain

2.1 Ease of Access vs. Abstraction

Much of the type information which the compiler deals with is not composed of types proper (int, $bool \rightarrow bool$), rather, it's in the form of "type constructors". Constructors create a language for computing types. These constructors are represented as an ML datatype in the compiler source code.

Datatypes very conveniently allow us to describe and operate on recursive structures (like type constructors). Unfortunately, by using a datatype to implement a data structure we are *committing to a representation.* We are not allowed to make the implementation choices we would be afforded when writing an abstract data type.

If we decide to implement type constructors as an ADT [3], then we are able to make representation choices but lose the ability to pattern match. Code interfacing with the ADT is stylistically less appealing and we lose some compile-time checking (such as checks for redundant or nonexhaustive matches).

Wadler [4] proposes a language mechanism, called *views*, for pattern matching against abstract data types. This feature allows the programmer to present an isomorphism between an arbitrary, abstract type and datatype-like constructors and destructors. With views, we have nearly the best of both worlds: pattern matching and flexibility in implementation. Unfortunately, *views are not in Standard ML!*

A compromise is available, though. We can present an interface to an abstract data type which gives us shallow pattern matching with minimal overhead. This interface is called the curtain, as in, "pay no attention to the code behind the curtain."

2.2 A Simple Curtain

The type constructor language in TILT has more than 10 different constructors, making it unwieldly for examples. Instead, I present the concepts in this paper using the lambda calculus. This is reasonably faithful to our constructor language, since it is a recursive datatype with a binding construct.

A typical implementation of the lambda calculus would be:

```
datatype exp =
     Var of string
   | App of exp * exp
   | Lambda of string * exp
```

As an abstract data type instead, the signature might look like this:

```
signature LAMBDA_ADT =
sig
     type exp

     val mkvar : string -> exp
     val mkapp : exp * exp -> exp
     val mklam : string * exp -> exp

     val isvar : exp -> bool
     val isapp : exp -> bool
     val islam : exp -> bool

     exception Wrong

     val getvar : exp -> string
     val getapp : exp -> exp * exp
     val getlam : exp -> string * exp
end
```

It is easy to see why this is unsavory. Constructing *exp*s is easy, but to manipulate them, we must use a series of nested `if` expressions. We also run the risk of an exception at run-time if we call the wrong `get` function.

Finally, here is a curtain interface for the same structure:

```
signature LAMBDA_CURTAIN =
sig
    type exp

    datatype exp_ =
         Var of string
       | App of exp * exp
       | Lambda of string * exp

   val expose : exp -> exp_
   val hide   : exp_ -> exp
end
```

The abstract type *exp* is the real representation of expressions "behind the curtain". Constructing *exp*s is easy:

```
val w = hide (Lambda("x", hide (App(hide (Var "x", Var "x")))))
```

Note that we always `hide` after applying a constructor. In a full-featured curtain interface we include functions corresponding to each constructor which also `hide` (ie., `hide o Lambda`). This means that constructing *exp*s is exactly as it would be with standard datatype constructors.

Manipulating *exp*s is almost as simple. Here's a function which converts lambda expressions to strings, using the curtain:

```
(* tostr : exp -> string *)
fun tostr exp =
   case expose exp of
       Var v => v
     | App (e1, e2) => "App(" ^ tostr e1 ^ ", " ^ tostr e2 ^ ")"
     | Lambda (v, e) => "\" ^ v ^ "." ^ tostr e
```

Aside from two important differences, this code is written the same way as it would be if we were using the datatype version. First, we must `expose exp`, because it is abstract (so we must write our functions using a `case` expression rather than the equivalent clausal function declaration syntax). Second, the *exp_* type is only exposed to one level, so we are constrained to patterns that are only one level deep. Deeper patterns need to be expanded:

```
(* traditional datatype implementation *)
fun eta exp =
   case exp of
       Lambda(v, App(e, Var v')) => if v = v' andalso
                                       not (free v e)
                                    then e
                                    else exp
     | other => other
```

Becomes:

```
(* curtain implementation *)
fun eta exp =
   case expose exp of
       Lambda(v, e) =>
          (case expose e of
               App(e, e') =>
                 (case expose e' of
                      Var v' => if v = v' andalso
                                   not (free v e)
                                then e
                                else exp
                    | _ => exp)
             | _ => exp)
     | other => exp
```

Note that we needed to repeat the expression in the default several times. It is possible to construct patterns where not just the expression is repeated, but the result of a non-trivial transformation:

```
(* traditional datatype implementation *)
fun f exp =
   case exp of
       App (x, Lambda(v, Var v')) => 1
     | App (Var v, y) => 2
     | App (x, y) => 3
```

```
(* curtain implementation *)
fun f exp =
   case expose exp of
       App (x, y) =>
          (case expose y of
               Lambda (v, y') =>
                  (case expose y' of
                       Var v' => 1
                     | _ => (case expose a of
                                  Var v => 2
                                | _ => 3))
             | _ => (case expose a of
                          Var v => 2
                        | _ => 3))
```

This second one requires pattern duplication because its clauses are not *disjoint*, and therefore the order in which they appear matters. This requires duplicating both code *and* the result of translating smaller patterns. This can translation can be tedious for a human to perform, and can be difficult to maintain. However, both would be easily implemented as mechanical program transformations, since they mirror the well-understood problem of pattern compilation [5].

Fortunately, empirical and anecdotal evidence both suggest that this kind of deep pattern matching is rare. For instance, when converting approximately the 10,000 lines typechecker and support code in TILT, we only needed about 10 non-trivially reworked patterns because of the curtain's limited depth. Qualitatively, patching an existing system to use the curtain interface is a tedious but not difficult job. Writing new code which uses the interface is generally no more difficult or tedious than writing standard pattern-matching code.

The most important feature of the curtain is that the *exp* type is abstract, and that we are now able to do arbitrary operations in the `expose` and `hide` functions. Some operations germane to TILT are given in the next section on wizards. One can imagine other uses for the curtain as well, such as creating lazy or infinite data structures which still allow pattern-matching. For instance, it's very easy to provide the ability to make lazy *exp*s which work transparently with the rest of the code; they are computed by the `expose` function when needed.

2.3 Alternate Curtain Interfaces

One possible problem with the curtain is that our `hide` and `expose` functions work on only a single level at a time. This somewhat constrains the range of operations we can perform.

Peter Lee suggests an interesting variant on the curtain proposed in the previous section. We can generalize the "carrier" datatype by making it polymorphic:

```
signature LAMBDA_CARRIER =
sig
    type exp

    datatype 'a carrier =
         Var of string
       | App of 'a * 'a
       | Lambda of string * 'a

   val expose : exp -> exp carrier
   val hide   : exp carrier -> exp
end
```

This has equivalent functionality to the curtain interface proposed above. However, if we have a special optimization which we can do with two levels available, we can add a new hiding function:

```
val hide2 : exp carrier carrier -> exp
```

Or perhaps we want to allow arbitrarily many `carrier` levels:

```
datatype multi_exp =
      More of multi_exp carrier
    | Done of exp carrier

val hide_deep : multi_exp -> exp
```

This all works out, though in our particular case there didn't seem to be any compelling use for this increased granularity. In other situations, this interface may be preferable to the less general one from the previous section.

3 The Wizards

The primary goal of the curtain is to allow us to use a clever implementation strategy, and moreover, to allow us to interchange clever implementations without affecting the client code. Therefore, we are mostly concerned with what goes behind the curtain, which we call the Wizard.

This section covers a few wizards; some we tried and some we did not. In order to measure the performance of these (and to gauge the difficulty of converting existing code to use the curtain), we instrumented a copy of the Intermediate Language's typechecker to use the curtain interface. The instrumented type checker coexisted with the old type checker, so that we could compare performance between them.

3.1 Null Wizard

A natural experiment to try is the Null Wizard, where the exposed and hidden types are the same:

```
structure NullWizard :> LAMBDA_CURTAIN =
struct
    datatype exp_ =
        Var of string
      | App of exp * exp
      | Lambda of string * exp

    withtype exp = exp_

    fun expose x = x
    fun hide x = x
end
```

In other words, this implementation is equivalent to the standard datatype approach, except that we make some extra calls to the functions `expose` and `hide`, which do nothing.

Disappointingly, code using the Null Wizard is 20% slower than plain datatype code. This is true both of code generated by SML/NJ and TILT itself. It is somewhat surprising that SML/NJ apparently chose not to inline (and then partially evaluate-away) calls to `expose` and `hide`. We expect that real wizards will not have inlinable `expose`/`hide` functions, so this is a fair estimate of the function call overhead.

The implication here is that it costs us 20% up front, just to use the curtain interface at all. This is not great, but it is not disaster. We suppose that by using the Wizard to reduce the heap size, we can reduce the number of swaps that TILT has to make, which could dramatically speed up compilation…

3.2 Exploiting Sharing

Reducing the heap size seemed to be the most promising optimization, since TILT uses a lot of memory and often needs to resort to swap space on disk, which is very slow. One straightforward way of reducing the heap size is by sharing memory more efficiently.

Since our type data structures are immutable, it's acceptable (and desirable) for types to share memory. For instance,

```
App(Var "x", Lambda ("y", Var "y"))
```

and

```
App(Lambda ("y", Var "y"), App (Var "z", Var "u"))
```

can share exactly the same memory for the $\lambda y.y$ part. All of the `Var "y"` expressions (and the strings `"y"`) can share memory as well. Thus the same data can be stored with less memory and fewer garbage collections. Sharing also improves performance by increasing memory locality and thus cache efficiency.

Some amount of deliberate sharing can be created through careful programming; TILT does this now. I refer to this as "static" sharing. In highly-redundant structures like types, though, lots of sharing is possible in ways unanticipated by the programmer because of the way the program behaves at run-time. Static sharing is also often difficult to maintain; operations like substitution can unnecessarily break apart shared nodes if not written carefully.

3.2.1 Hash Consing

Hash consing [6] is a technique for exploiting sharing at runtime. Allocating a datatype constructor in SML is often called *consing*, after the related activity in lisp implementations. The idea of hash consing is that instead of always allocating a new constructor (or "cons cell"), we first consult a hash table to see if we've already allocated an identical constructor. If so, we reuse the existing memory. I call this kind of sharing "dynamic" sharing.

There are lots of issues complicating hash consing, including garbage collection (our table must eventually disconnect unused data so that their memory can be reclaimed). We've also experimented using alternate lambda encodings [7] to share some terms which are semantically (but not structurally) equivalent. See the section on de Bruijn indices for why this turned out to be a bad idea.

3.2.2 Implementing Hash Consing

In order to efficiently perform hash consing, we need to add some overhead to our representation.

```
structure HashWiz :> LAMBDA_CURTAIN =
struct
    datatype exp_ =
        Var of string
      | App of exp * exp
      | Lambda of string * exp

    withtype exp = { e : exp_, hashcode : word, stamp : stamp }

    fun expose { e, ... } = e
    fun hide x = ...
end
```

Now our abstract *exp* type has three components: `e`, which is the term, `hashcode`, which is the cached hash code for the term, and `stamp`, which is a unique stamp for expression.

Our hide function (not shown) has the following steps. First, calculate the hash code for the new cons cell. Because we cache the hash code of all its child nodes, this can be done in constant time.[1] We then check through our hash table, looking for nodes that are equivalent to the target. If we find any, we return that old node. Otherwise, we insert the new node into the table along with a unique *stamp*.

This search requires a fast check for equality between constructors. Fortunately, caching the hash code give us a quick way to decide: First, check the stamps. If the stamps are the same, then the types are certainly the same. Next, check the hash code. If the hash codes are different, the types are certainly different (though our type equality rules might later conclude that the types are *equivalent*). If the hash codes are the same, we recursively check subterms of the types using the same method.

It is worth being explicit about the overhead here: Each constructor node has an extra 2-3 words (depending on the compiler's representation) stored with it. It is also important to note that we do not reduce the number of allocations performed (we need to create the thing that we want, so that we know what to look for); we only potentially reduce the working set. In fact, hash consing requires significantly more allocations, though the idea is that most of them are very short-lived. As mentioned earlier, however, reducing the working set might reasonably be expected to improve performance (because of memory locality and reduced swaps) despite these costs. Unfortunately, we will see that the gains are rather modest compared to the overhead.

3.2.3 Hash Table Collection Scheme

As we insert nodes into the hash table, the table fills up. Since a typical compilation session might involve the creation of hundreds of thousands of types, we need some way to remove unneeded types from the table. In order to approximate evicting the least desirable types (ones that we think we're least likely to reuse), we use a Least Recently Used scheme. When a node is first inserted, or when it is shared, it is moved to the front of a queue. When our hash table is filled to its limit, we remove types from the end of the queue. Therefore, types that were seldom shared and that were created long ago are removed first.

The results for the LRU scheme are very encouraging. Figure 1 is a graph of table size (number of elements allowed in the hash table) versus the percent of shared nodes. Note that this is a logarithmic graph, and that hash consing still gets dramatic sharing even with a modest table size ($2^{11} = 2048$ elements)! These figures indicate very high (temporal) locality in the types we create in the TILT IL. One interesting result here is that we 11% of the time we create a type constructor, we can share with the *last constructor created* (LRU cache size of 1)! This may just be an artifact of the somewhat peculiar behavior of a typechecker (the

[1] In the actual TILT type constructor language, many constructors have lists of arbitrary length. The time to compute the hash code is linear in the length of these lists, but independent of the depth of the subtrees.

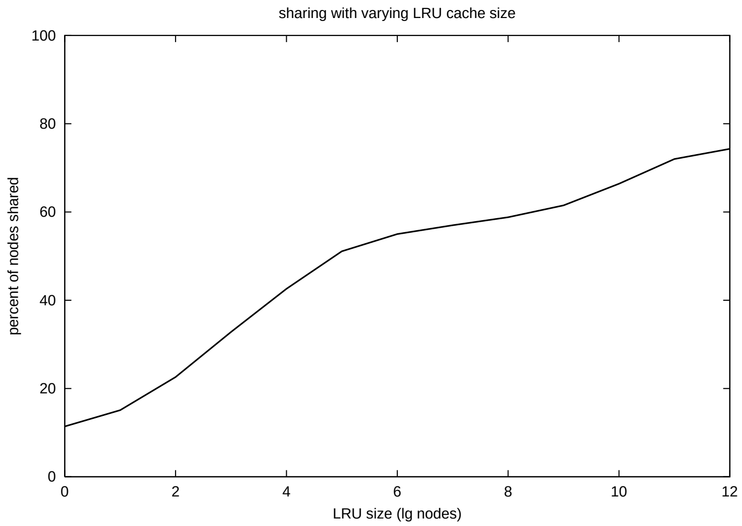

Figure 1: Logarithmic comparison of LRU size to percent sharing

common case is to be checking equality between two equal types), but is still surprising.

3.2.4 de Bruijn Indices

One wizard we implemented used de Bruijn-encoded lambda constructors internally in an attempt to increase the likelihood of sharing when hash consing.

The de Bruijn encoding [4] uses integers ("de Bruijn indices") instead of named variables to mark the correspondence between a bound variable and its binding site. For instance, the terms $\lambda x.x$ and $\lambda y.y$ are both represented as $\lambda.\#0$ (both x and y are bound at 0 levels deep). $\lambda x.\lambda y.xy$ would be $\lambda.\lambda.\#1\#0$ with de Bruijn indexing.

Since the names of bound variables disappear, the expectation is that constructors are more frequently represented in the same way (and so they can share memory more often). Unfortunately, de Bruijn-indexed terms are clumsy to manipulate. For instance, substituting into a de Bruijn term requires that free indices in the substituted term be renumbered, because they're now at a different depth. de Bruijn indices are also much more difficult to program with. Because of this, if we were going to use de Bruijn encoding, we wanted to do so only in the wizard internals.

Here's how it worked: When a node with a binding site (like λ) is pushed behind the curtain with the `hide` function, it translates all references to that bound variable (within its subterms) into de Bruijn indices. We call this operation "binding". When a binding site is exposed to the client, the wizard generates a brand new variable name and renames all the corresponding indices. This operation is "dubbing". Client code is expected to respect alpha-equivalence, so changing the names of bound variables is legal. For example (datatypes starting with W are abstract and not visible to the client):

Client code	Real value
`val t1 = hide (Var "x")`	`WVar "x"`
`val t2 = hide (Var "y")`	`WVar "y"`
`val t3 = hide (App (t2, t1))`	`WApp(WVar "y", WVar "x")`
`val t4 = hide (Lambda ("x", t3))`	`WLambda(WApp(WVar "y", WIVar 0))`
`val o1 = expose t4`	`Lambda ("newvar_100", WApp(WVar "y", WVar "newvar_100"))`

It's important to take note of the sequence in which terms are created. Though we may read $\lambda x.x$ from left to right and think of the λ part "happening first", datatypes are created from the inside out. x is a free variable until it is bound by enclosing it in a λ.

The "bind" and "dub" operations used to maintain the illusion of named variables outside the curtain are somewhat expensive. We hoped to make up for this expense with increased sharing, as de Bruijn terms seemed like they would be more prone to dynamic sharing than named ones. We also intended to defray the cost of doing these operations with tricks like laziness and memoization.

One of the optimizations we implemented associated a cost with each free variable that corresponded to its depth and frequency within the term. We also allowed the possibility of using either named variables or de Bruijn indices for each binding site. When the variable is finally bound, we can decide whether to use de Bruijn indices (if the cost of binding is acceptable), or retain named variables (if the cost is too high).

To decide an appropriate threshold, we compiled the Standard ML Basis Library with TILT at settings ranging from 0 (only use de Bruijn indices if the cost is less than 0; that is, never) to infinity (always use de Bruijn indices). The cost was never more than about 50, so a threshold of 50 is a suitable "infinity" for this experiment. For each run, we record the number of *shared* constructors (found in the hash table) and *new* constructors created (entries added to the hash table). The ratio of shared constructors to the total (shared + new) is the percentage reuse. We also estimate the size of these nodes in machine words, and provide weighted measurements of shared memory and new memory. The results are surprising (Figures 2, 3).

In Figure 2 we graph the percentage of total constructors shared (and a separate line weighted by the size of the constructor nodes) versus the bind threshold. At 0 (never binding), approximately 75% of nodes created were found in the hash table and shared. Weighted by the size of the nodes, we use approximately 65% of the memory we would normally use without hash consing. These numbers shoot up to almost 90% and 82%, respectively, for a bind threshold of 1. A threshold of 1 corresponds to using de Bruijn indices only when they cost nothing: when the bound variable isn't ever used in the term. This is a dramatic improvement. Unfortunately, once we start actually doing bind and dub operations at (thresholds 2 and higher), the sharing percentages dip disappointingly and even out around 60% and 50%. The reduced effectiveness is an unfortunate (and subtle) side effect of the internal work done renaming variables and creating new terms with de Bruijn indices. The cost of internal work is shown better in Figure 3, which shows the approximate number of words allocated and shared for each binding threshold. Never using

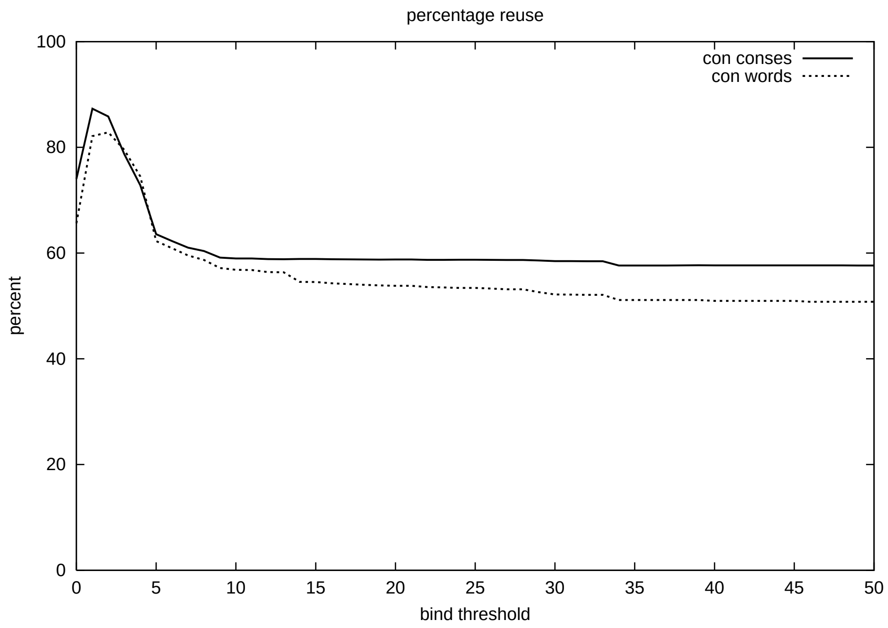

Figure 2: Percentage reuse (counting internal operations)

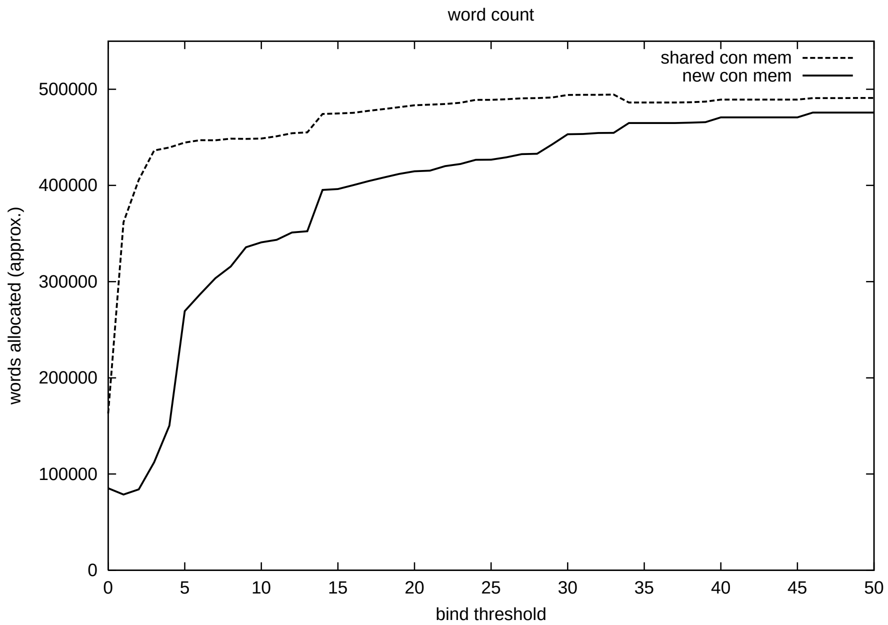

Figure 3: Memory allocated/shared (counting internal operations)

de Bruijn indices, we allocate about 75,000 words and share 160,000. At a threshold of 1, we allocate slightly fewer words and share more than twice as many as before. For higher thresholds, the amount of shared memory increases significantly, but the number of new memory words allocated increases as well. Only at a threshold of 1 have de Bruijn indices actually helped to reduce our working set.

Based on these results we decided *not* to use de Bruijn indices except in the degenerate case where the bound variable was never used (the peak of the curve in Figure 2 and lowest point in Figure 3). This means that $\lambda x.y$ and $\lambda z.y$ are both represented as $\lambda.y$, but $\lambda x.x$ and $\lambda y.y$ remain as they are (and do not share memory). This way we can avoid bind and dub operations altogether: We need only to erase the variable name from binding sites whose variable is never used, and generate a new name each time one of these anonymous binding sites is exposed. de Bruijn indices #0, #1, etc. never appear.

It's easy to make the mistake of thinking de Bruijn-encoded terms are universally better at sharing than terms with named variables. Though the de Bruijn encoding has the advantage of alpha-equivalent terms being represented in the same way, some named terms can have more sharing possible. For instance, $\lambda x.(xx)(\lambda y.(xx))$ can share the two applications (xx) as well as all of the x variables. Represented as a de Bruijn term, however, allows for less sharing: $\lambda.(\#0\#0)\lambda.(\#1\#1)$. The two #0 indices can be shared, as can the two #1. But the applications are of different subterms, and cannot be shared as in the named version.

Sharing in de Bruijn terms versus named terms corresponds to an idea of "shape pervasiveness" versus "name pervasiveness". It's difficult to gauge which is more effective for representing types during the phases of a compiler. Our experiment provides evidence, at least, that mixing the two does not appear to be efficient enough to be effective.

4 Too Much Overhead

From the measurements given in the previous sections, it is clear that there exists a substantial amount of potential dynamic sharing available to a hash-consing Wizard. Unfortunately, our wizards incur far too much overhead in harnessing this sharing.

In figures 4 and 5, we show the performance of a few wizards compiling the basis library on two different machines. The results are catastrophic: the type-checker is universally many times slower than it is without the curtain or wizards.

Basline is the old code, which uses a datatype to represent type constructors. **Null Wizard** is as described above; hide and expose are the identity. **WFV** is the wizard which keeps track of free variables so that it can drop unused bindings. **WFVU** is the same, using unsafe (not bounds-checked) array operations. **Hash** is the wizard which only hash-conses; it does not drop unused bindings. **Monolith Null** and **Monolith FV** are the same as **Null Wizard** and **WFV** respectively, but are built as one compilation unit to stimulate inlining.

All timings were done using TILT built with SML/NJ, though the results for TILT built with TILT show about the same ratios. Times reflect wall-clock times (CPU times are similar, but do not reflect savings from fewer swaps).

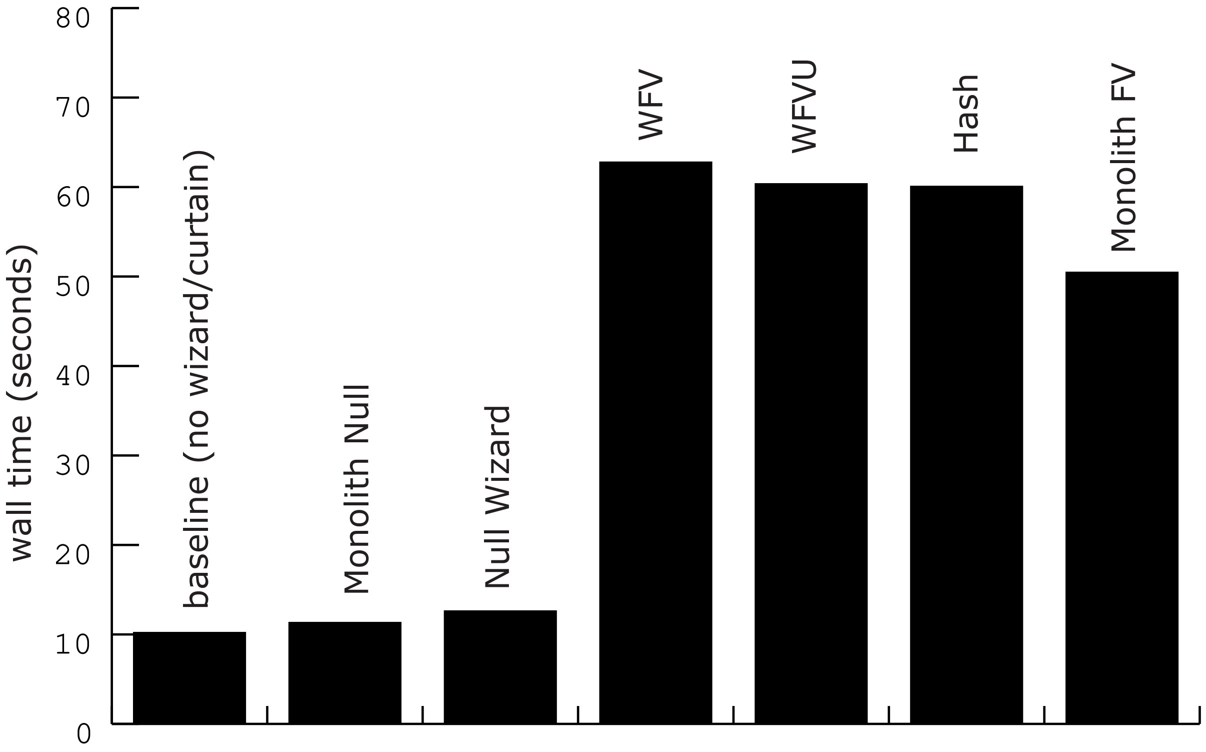

Figure 4: Wizard Performance, Ram=1536 Mb

One possible explanation is that we are not actually saving memory. Recall that in order to support hash-consing, we need at a minimum 2-3 extra words per node (for one wizard, we also store a memoized set of its free variables, which could potentially be very large). From the results in figures 4 and 5, it seems that we are do have a modestly more compact working set (the "low memory" machine is about 4.1 times slower, while the "high memory" machine is about 6.2 times slower). It's clear that the savings are not dramatic enough to make up for the overhead, though.

The garbage collector also tends to have difficulty with the hash table. Though garbage collection times are small in general, they rise rapidly as we increase the size of the hash table. This is probably explained by the assumptions made by a generational collector. A generational garbage collector expects that items in old generations are infrequently updated to point to items in new generations, but this is exactly what the hash-conser does: it makes the hash table (whose lifetime is the same as the program's) point to types which were just created.

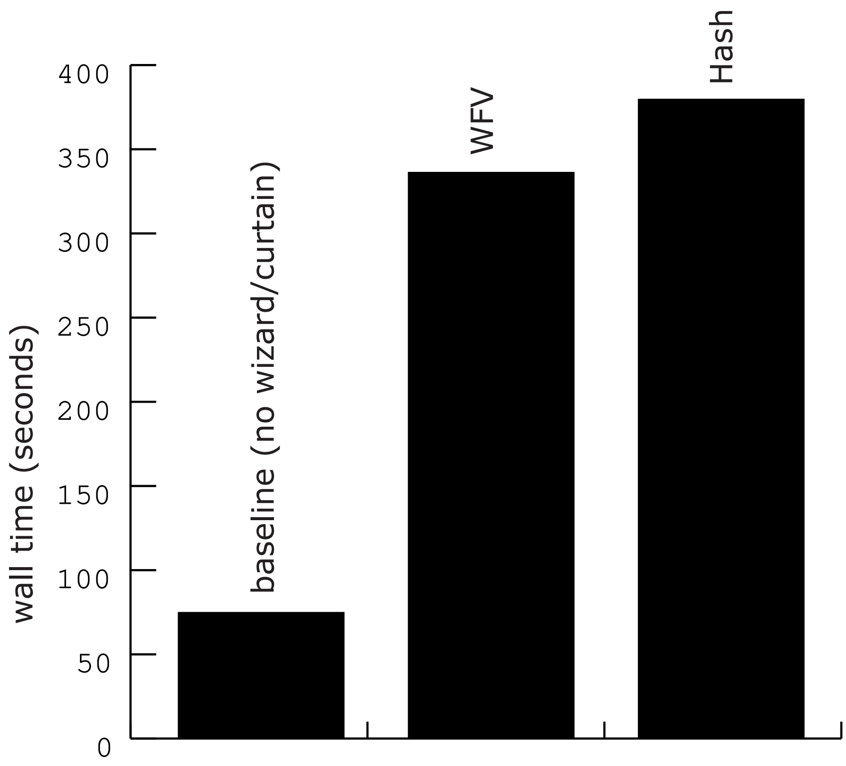

Figure 5: Wizard Performance, Ram=96 Mb

4.1 Where is the time spent?

In addition to the memory overhead, we have a significant cost in maintaining the hash table and finding nodes with which to share. Here are the results of profiling the Hashing wizard on a compilation of the basis library:

%	sec.	procedure	description
0.7	14.1787	WizhashfnLRU.f	Search and maintenance of hash table
0.5	9.7471	HashWiz.cdt_equal	Test if the outermost level of two constructors is equal
0.5	9.7041	HashWiz.cexpose	Expose a constructor
0.5	8.8281	WizhashfnLRU.share	Search and mainteance of hash table
0.3	5.8496	HashWiz.c_equal	Test if two constructors are equal (by comparing their stamps and hashcodes, or calling cdt_equal)
0.3	5.7881	HashWiz.chide	Hide a constructor
0.3	5.3818	DLL.disconnect	Maintenance of LRU cache
0.2	4.6377	HashWiz.cmaker	Generate stamp, and try to share a new constructor
0.2	4.3535	HashWiz.hashcode	Calculate hash codes
0.2	3.8359	WizhashfnLRU.loop	Remove entries when the hash table is full
0.2	3.4268	DLL.move_to_tail	Maintenance of LRU cache
0.2	3.1758	DLL.add_tail	Maintenance of LRU cache
0.1	2.8193	WizhashfnLRU.f.inner	Remove an entry from the hash table
0.1	2.7305	HashWiz.kexpose	Expose a kind
0.1	2.1396	DLL.-->.inner	LRU utility function
		many more at 0.1% and below...	

The first column lists the percent of total compilation time spent in this function, and the second the number of seconds spent. These functions are ranked by the time spent within them, and only functions related to the wizard code are listed.

It's easy to see that hashing is expensive; `WizhashfnLRU.f` is the 20^{th} most "popular" function in the compiler, with over .7% of the total execution time spent in it. However, it's not that these functions are particularly inefficient—just that they are called many, many times. For instance, the highly-ranked `HashWiz.c_equal` function is only one line long: it compares two pairs of `word`s, then calls `cdt_equal`. Many other functions are similarly brief, such as most of the LRU maintenance functions.

The key fact is that the IL typechecker spends almost all of its time constructing and destructing datatypes. Replacing this with functions which do even a small amount of work is devastating to its performance.

5 Related Work

5.1 SML/NJ

Appel and Gonçalves [8] implemented hash consing as part of the garbage collector for an early version of SML/NJ. Their hash-conser works by performing the hash operation only when an object is promoted from the new generation to the old. By doing so, they avoid calculating hash codes for short-lived objects.

They acknowledge that the overhead for doing hash consing may be more than can be recouped through improving memory locality and shrinking the working set. Indeed, many of their benchmarks run slower with hash-consing turned on, and the best ones only improve execution time by 10%.

5.2 FLINT

The FLINT project at Yale uses hash consing, memoization, and de Bruijn indices to improve the performance of their intermediate language for the SML/NJ compiler. Their results show that these changes made compilation much faster [3]!

How then, do we account for the drastically differing results between the superficially similar TILT IL and FLINT implementations?

5.2.1 Asymptotic Complexity

The most fundamental reason why these implementations are incomparable is that the FLINT language lacks a constructor-level `let` construct. In ML, it is easy to create types which are exponentially large with regard to the size of the program. For example, they give a program similar to the following:

```
fun I x = x

val y = I I I ... I I 0
```

The rightmost `I` function (with polymorphic type $\forall t.t \rightarrow t$) is instantiated at `int`. The next `I` is instantiated at `int` $\rightarrow$ `int`, the next at `(int` $\rightarrow$ `int)` $\rightarrow$ `(int` $\rightarrow$ `int)`, etc. We get an exponential blow-up in the size of the types very easily, and a small number of `I` functions can quickly cripple a compiler which is implemented naively. A smart implementation will somehow share the left- and right-hand sides of each function type, so that the whole type takes only linear space.

FLINT achieves this asymptotic improvement in space usage with hash-consing. Additionally, since traversing this DAG would take exponential time even if stored in linear space, they provide a memoizing `fold` function which allows them to traverse it efficiently. Efficient implementation of this memoization also requires that they hash-cons identical nodes.

The TILT IL, on the other hand, has a constructor-level `let` construct. We can represent the sharing explicitly using `let`:

```
let t = let t' = let t'' = ...
                 in t'' -> t''
                 end
        in t' -> 't
        end
in t -> t
end
```

This allows us to represent the type in linear space and traverse it in linear time. Therefore, TILT already has much of the benefit that FLINT gets through hash consing in its careful use of the `let` constructor. Since we already take care of the asymptotic complexity, hash consing for TILT can only hope to provide a constant factor of improvement.

5.2.2 Abstraction

The FLINT interface (given as `LAMBDA_ADT` in 3.2) has two efficiency advantages over the curtain interface. First, no extra allocation takes place to present the client with a pattern-matchable datatype; the client instead must test the abstract type using the predicate functions and then call the appropriate destructor. Second, FLINT exposes de Bruijn indices to the rest of the compiler, which means that the expensive *bind* and *dub* operations are unnecessary. Of course, there is a tradeoff between efficiency and convenience here.

5.2.3 Other Differences

There are a number of other differences that are worth mentioning.

FLINT guarantees that that equivalent types share the same memory, so that type equality is just pointer equality. In our implementation, we support a fast "yes" (stamps are equal) and "no" (hash codes are different), but we do not guarantee that equivalent types have equivalent stamps. This choice gives us more flexibility in our hash table collection scheme, among other things.

FLINT memoizes some information about each term, such as its set of free variables, and its normal form. For our wizard which dropped unused bindings, we also memoized the set of free variables. However, normalization in our intermediate language is done with respect to a context [9]; there is no single normalized form for us to memoize!

6 Conclusions

From an engineering perspective, the curtain is successful. Without it, a large amount of code would have needed to be rewritten in order to attempt these experiments. It is reasonably simple to instrument existing code to use the curtain interface, and then very easy to try out many different wizards behind the curtain.

The curtain code may still be useful for other kinds of experiments. However, it is unlikely to be practical for performance-critical code unless the wizard behind it can recoup the 20% baseline overhead *just to use the curtain*, as well as its own implementation overhead. This may be a property of SML/NJ and TILT, or it may be because of the particular behavior of the IL typechecker. It is likely that other situations could use a curtain interface at a much smaller cost.

We learned that the performance of our typechecker is closely bound to the efficiency of datatype construction and destruction. Optimizing compilers for SML are good at optimizing datatype manipulation, so simulating datatypes through other means is costly. Leaner code appears to win out over clever code when the program is this datatype-heavy.

Contrary to our belief when starting this project, hash consing does not appear to be an appropriate way of gaining constant-factor improvements. It seems that it is best used to create invariants about a program (such as "all equivalent constructors are shared") or to make things like memoization more effective or equality more efficient.

Therefore, though significant sharing is definitely available in the TILT IL, it may be impossible to efficiently capture it dynamically. Perhaps this means that, in order to improve memory usage, we need to make better use of static sharing and our constructor-level `let`.

References

[1] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[2] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[3] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *International Conference on Functional Programming*, pages 313–323, 1998.

[4] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312. Association for Computing Machinery, 1987.

[5] P. Wadler. Efficient compilation of pattern matching, 1987.

[6] A. P. Ershov. On programming of arithmetic operations. 1(8):3–6, 1958.

[7] N. de Bruijn. A survey of the project automath, 1980.

[8] Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, 1993.

[9] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Carnegie Mellon University, 2000.