Performance Polymorphism

Ian Toyn, Alan Dix and Colin Runciman
Dept. of Computer Science, University of York
Heslington, York, YO1 5DD, UK

Abstract

In an interactive functional programming environment with a Milner-style polymorphic type system (Milner 1978), a modification to one definition may imply changes in the types of other definitions. A polymorphic typechecker must carry out some *re-typechecking* to determine all of these changes. This paper presents a new typechecking algorithm which performs fine-grained re-typechecking based on analysis of individual type constraints. The new algorithm is compared with that of Nikhil (Nikhil 1985), which performs re-typechecking of entire definitions.

1 Background and Motivation

The first author has constructed a programming environment, called *glide*, to support exploratory programming in a purely functional lazily-evaluated language. The glide language is similar to Miranda (Turner 1985). A glide program is a collection of definitions and an expression using them. Programming in glide involves introducing new definitions, performing computations involving the current definitions, and modifying definitions. The novel features of the glide environment include its debugging tools and its polymorphic typechecker. The debugging tools have been described elsewhere (Toyn 1986). The typechecker imposes a Milner-style polymorphic type system (Milner 1978) without introducing long delays that would disrupt exploratory programming.

In *Practical Polymorphism* (Nikhil 1985), Nikhil presents an algorithm that addresses the re-typechecking problem. His algorithm is based on the observation that if a definition does not use a modified definition, either directly or indirectly, then its type cannot be affected by changes in the type of the modified definition. This algorithm identifies those definitions whose types might change from the dependency graph of the program. Only the identified definitions are re-typechecked.

An early version of glide used Nikhil's re-typechecking algorithm. However, an automatic loading mechanism was subsequently designed for glide which interacted badly with Nikhil's algorithm. In each session, needed definitions have to be loaded from the file system into the programming environment before they can be used. As each definition is loaded, some previously loaded definitions may have to be re-typechecked. To minimize re-typechecking, it is best to load definitions at the leaves of the dependency graph before those higher up the graph. But glide's automatic loading mechanism causes definitions to be loaded only when they are first needed in a computation, that is from the root of the dependency graph down towards the leaves. So worst-case behaviour was always obtained from the re-typechecking algorithm. Rather than change the loading mechanism, the problem was solved by an improved re-typechecking algorithm as described here.

Summary of Contents

A new typechecking algorithm is first developed for a grossly simplified language without recursive and local definitions. Having argued the correctness of this algorithm, more efficient algorithms are developed, still for the simplified language. Only then is the most efficient algorithm extended to cope with recursive and local definitions.

Terminology

It is assumed that the reader is familiar with basic typechecking and unification terminology, for example *type variable*, *substitution*, and *genericity* (Cardelli 1985). The notation `@a`, `@b`, `@c`... is used to denote type variables. The following additional terminology is needed to discuss re-typechecking.

An identifier is either *free* or *bound*. A free identifier is a use of a definition. A bound identifier is a use of a function's parameter. The *parent definition* of an identifier is the closest definition enclosing that identifier.

Each atomic expression (that is each identifier or literal), `a`, in an expression has a corresponding *type constraint* denoted `(a)` of the form

$$E_a = I_a \qquad (a)$$

where E_a is the *expected type* and I_a is the *inferred type*. The expected type E_a is determined from the context of the atom within the entire expression. The inferred type I_a is determined from the atom itself. There is one type constraint `(a)` for each static occurrence of `a`. (The examples will avoid expressions containing more than one occurrence of the same atomic expression, and so this notation will suffice without ambiguity.)

Unification of a type constraint in the context of a set of substitutions extends that set of substitutions so as to constrain the expected and inferred types to be equal. The operation of determining the value of a type constrained by a set of substitutions is known as *pruning*.

A *free identifier's inferred type* is (a copy of) the inferred type of the identifier's definition.

Re-typechecking a free identifier `i` involves *re-unifying* the (possibly revised) type constraint `(i)`.

Note

The algorithms are presented on the assumption that the name of a definition is a single identifier. If the name is a pattern involving several identifiers, some explanations would become clumsy: any reference to uses of a definition may be read as uses of the definition via any of these identifiers, and the inferred type of a definition should be re-expressed as inferred types for each identifier in its name. These abbreviations allow patterns to be ignored throughout the remainder of this paper.

2 The Initial Idea

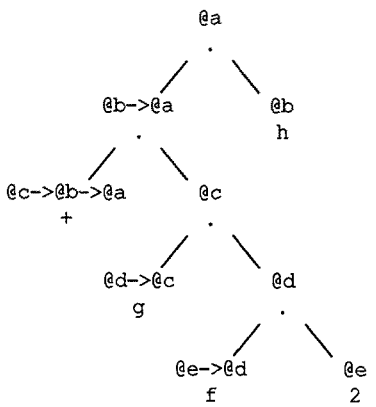
The initial idea arose from the realization that re-typechecking entire definitions is excessive: *most of the type constraints remain the same.* All the algorithms to be presented confine re-typechecking to bindings between definitions rather than entire definitions.

Example

Consider the following definition. (This example will be returned to repeatedly as successive algorithms are developed.)

```
Define def -> g (f 2) + h
```

In preparation for enumerating all type constraints involved in typechecking this definition, introduce expected types for each component of the definition, based on its applicative structure.



Now enumerate the type constraints. Each has the expected type on the left-hand side, and the inferred type on the right-hand side.

$$\texttt{@c -> @b -> @a} = \texttt{num -> num -> num} \quad (+)$$

$$\texttt{@d -> @c} = I_g \quad (g)$$

$$\texttt{@e -> @d} = I_f \quad (f)$$

$$\texttt{@e} = \texttt{num} \quad (2)$$

$$\texttt{@b} = I_h \quad (h)$$

Note that both E_f and E_g use the type variable @d. This expresses the requirement that the result type of I_f must unify with the parameter type of I_g. When one of the type constraints (f) and (g) is unified, any resulting substitution for @d will refine the expected type of the other. If the inferred type of the one changes, the refinement may change, in which case the unification of the other must be repeated. So re-typechecking only the free identifier whose definition has changed type is sometimes insufficient.

This example also illustrates the fact that it is unnecessary for the entire definition to be re-typechecked: only I_f, I_g, and I_h can change (barring redefinition of primitives such as +), so the other type constraints, (+) and (2), need be unified only once *provided* they are unified first. (The need for this proviso will be explained later.)

One thing not illustrated by this example is that a change in the inferred type of a free identifier can cause a change in the type inferred for its parent definition. In this case, all uses of that definition must be re-typechecked. This refinement check is performed by Nikhil's re-typechecking algorithm.

3 Typechecking a Simple Language

Suppose there are no recursive definitions and no local definitions. These restrictions have the effect of avoiding type variables whose genericity changes during re-typechecking.

3.1 Outline of Algorithm A

Algorithm A consists of two phases. The first phase infers type constraints corresponding to literals, primitives, and bound identifiers. Only the type constraints corresponding to non-primitive free identifiers, whose definitions might change, are left to the second phase. The first phase is done immediately a definition is loaded. Although types are known for those free identifiers whose definitions have already been loaded, it will be shown that ignoring them until the second phase reduces the cost of re-typechecking.

The first phase may be thought of as typechecking a definition in isolation, unaware of the types of free identifiers' definitions. In all other respects it is exactly like Nikhil's "improved type-checker" that gives useful diagnostics of type errors. The difference amounts to having an initial environment that contains types of primitives alone. At the end of the first phase, an *approximate* type has been inferred for the definition and *approximate* expected types have been inferred for each of its free identifiers. These approximate types may be further refined by the second phase.

The second phase infers one type constraint for each free identifier, unifying its expected type with the type inferred from the identifier's definition. The second phase may be interleaved with actual computation. It can be done either as definitions of the free identifiers become available (eagerly) or only as they are required during computations (lazily). The second phase is required to backtrack should any of the free identifiers' definitions be changed. Figure 1 sketches a process for typechecking a definition, implementing the two-phase algorithm in which the second phase is done lazily. The auxiliary `idcheck` has been extracted so that it can be shared by the re-typechecking process.

```
idcheck i def ->
    unify E_i with I_i, extending def's substitution;
    flag (i) as having been unified.
```

```
typecheck def ->
  -- phase one
    Nikhil_typecheck def in a type environment
    initially containing only the primitives;
    prune I_def;
    for each free identifier, i, in def,
        prune E_i;
    forget all substitutions from phase one;
  -- phase two
    for each free identifier, j, in def,
        flag (j) as not unified;
    when the value of one of def's free identifiers, k, whose type
    constraint (k) has not been unified, is needed by a computation,
        idcheck k def;
    when the inferred type of one of def's free identifiers changes,
        forget all substitutions from phase two;
        start phase two afresh.
```

Figure 1: Algorithm A

The first **when** clause in Figure 1 may have the effect of refining the type of def, in which case the second **when** clause fires in the processes typechecking the definitions that use def. These instances of the second **when** clause may unrefine the type of their defs with similar effects. A pictorial view is useful in conceptualising the extent of these effects. The graph of typechecked uses between definitions is a directed acyclic graph (DAG), with a unique root at which is the user's expression to be evaluated. The first use of each free identifier in a computation corresponds to adding a new arc to the DAG, possibly to a new node. Changing a definition corresponds to removing a node and all arcs to or from it; any replacement definition will be linked into the DAG on its first use in a computation. If addition or removal of an arc refines the type of the definition at its source node, adjacent arcs on paths from that node back towards the root have to be re-typechecked. Algorithm A demands re-typechecking of every arc whose source is on these paths.

Re-typechecking

The arcs to be re-typechecked initially form a connected DAG within the complete DAG. They are re-typechecked by a single re-typechecking process. Re-typechecking an arc may reveal more information about the definition at the source of that arc, thus requiring more re-typechecking to be done higher up the DAG. Determination of obsolete arcs and re-typechecking of arcs are therefore interleaved. Re-typechecking should be done from the leaves towards the root, so that type information about a definition is inferred before re-typechecking uses of that definition. Otherwise some arcs may be re-typechecked more than once. Only when all obsolete arcs have been successfully re-typechecked (that is when the re-typechecking process has terminated) is the computation permitted to proceed. One re-typechecking process is created whenever an arc is added to or removed from the DAG. A re-typechecking algorithm that is suitable for use with Algorithms A–D is given in Figure 2. Its correctness and performance are discussed in Section 3.2.

```
while there remain free identifiers to be re-typechecked,
    let def be a definition of a free identifier to be re-typechecked,
    chosen such that def is not the parent definition of any free identifiers
    yet to be re-typechecked;
    for each free identifier, i, waiting to be
    re-typechecked whose definition is def,
        let p be i's parent definition;
        idcheck i p.
```

Figure 2: Re-typechecking (Algorithms A–D)

Example (continued)

For the example definition,

```
Define def -> g (f 2) + h
```

the first phase of the algorithm unifies both (+) and (2), and hence infers the following approximate expected types for the free identifiers,

E_f is num -> @d

E_g is @d -> num

E_h is num

and an approximate type for the entire definition.

I_{def} is num

Suppose that in the second phase a definition for g is found such that I_g is num -> num. This type is unified with E_g giving the singleton set of substitutions {num for @d}. E_f is refined to num -> num. Suppose definitions are then found for f such that I_f is num -> num and for h such that I_h is num. All free identifiers are typechecked without any type errors being found. Should I_f or I_g change, the other free identifier is re-typechecked, as required, by repeating all of the second phase. This is illustrated in the following DAG, in which broken lines indicate those free identifiers that are re-typechecked.

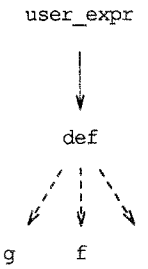


Figure 3: DAG of example for Algorithm A

Suppose it was I_f that changed. That explains why the arc from def to f is broken. The arc from user_expr to def is not broken because nothing could have happened in the second phase to refine I_{def}; paths of dependence lead towards the root, but need not go all the way. The arcs from def to g and h are broken because their source is the source of another broken arc. Note that the first phase, which unified both (+) and (2), is never repeated; this is a significant saving.

3.2 Correctness of Algorithm A

That Algorithm A computes the correct type for a definition may be shown by proving that it computes the same type as would be computed by Nikhil's algorithm.

Lemma 1. Algorithm A infers the same system of type constraints as Nikhil's algorithm.

Proof. The expected types of all type constraints are the same in the two algorithms, since they are generated by the same construction. The inferred types of literals and primitives are clearly the same. The inferred types of free identifiers will be the same exactly if Algorithm A is correct. This can be shown by induction, starting from those definitions that use no other definitions (those at the leaves of the DAG). Since all expected types and inferred types are the same, the type constraints must be the same. □

Lemma 2. Changing the order in which a system of type constraints are unified may result in a different set of substitutions, but pruning a type variable according to any of these sets of substitutions will always give the same type (up to renaming of synonymous type variables).

Proof. This follows from the associativity of unification (Robinson 1965). □

Theorem 1. Algorithm A infers the same type for a definition as Nikhil's algorithm.

Proof. Lemma 1 shows that the algorithms infer the same system of type constraints. They will almost certainly unify the constraints in different orders, but *Lemma 2* shows that they will nevertheless compute the same type. □

In dispensing with the set of substitutions resulting from the first phase, Algorithm A assumes that none of the work of the first phase need ever be repeated. The correctness of this may be argued as follows.

Theorem 2. The first phase of typechecking need never be repeated.

Proof. Whenever the type of a free identifier's definition changes, the corresponding type constraint and the set of substitutions resulting from that constraint's unification become obsolete. *Later* unifications of other type constraints in the context of these substitutions must be repeated. *Earlier* unifications of type constraints cannot depend on the now obsolete substitutions, and so need not be repeated. So, since the type constraints that can be made obsolete by changes in the inferred types of free identifiers are all unified in the *second* phase of typechecking, the *first* phase of typechecking need never be repeated. □

Never having to repeat the first phase is a considerable saving. It also allows the inferred type of the definition and the expected types of the free identifiers to be pruned and the substitutions forgotten at the end of the first phase, thus reducing the cost of subsequent pruning operations. Furthermore, each definition can have its own separate set of substitutions, since it shares no type variables with other definitions; so that forgetting substitutions simplifies to emptying a set.

Some more terminology will assist discussion of re-typechecking. Let the *depth* of a node in the DAG be the length of the longest path from that node back to the root, so that a node with a smaller depth than another is *higher* in the DAG. Similarly, a node with a larger depth is *lower* in the DAG.

Theorem 3. The re-typechecking process of Figure 2 terminates.

Proof. The arcs to be re-typechecked form a DAG within the complete DAG. Each iteration of the outer loop eliminates a leaf from this smaller DAG. It may also require further re-typechecking higher up the complete DAG, but this clearly cannot always be the case as there are only a finite number of definitions higher up the graph. So the elimination of one leaf on every iteration guarantees eventual termination. □

Theorem 4. No arc need be re-typechecked more than once.

Proof. Suppose that the typechecking and re-typechecking processes are synchronized so that the re-typechecking process waits before re-typechecking any arc until the typechecking processes can make no further progress. Suppose further that an arc is re-typechecked only when there are no lower arcs to be re-typechecked. So all additional arcs to be re-typechecked must be higher up the DAG than all arcs previously re-typechecked, and consequently no arc will be re-typechecked more than once. □

This is an attractive result, but one that is not exploited by the re-typechecking algorithm of Figure 2. That algorithm determines an order in which to re-typecheck arcs from their connectivity rather than from their relative depths. As long as the DAG of arcs to be re-typechecked is connected (as it is initially), the same performance will be obtained. If it becomes disconnected in such a way that one component lies beneath another, then the algorithm of Figure 2 may re-typecheck arcs in the higher component before those in the lower component. So additional re-typechecking above the lower component may imply repeated re-typechecking of arcs in the higher component. In the worst case, this algorithm will have cost quadratic in the depth of the lowest arc in the initial DAG of arcs to be re-typechecked. Its performance in general is not so bad, and may compare favourably with the cost of computing depths.

Theorem 5. The second phase of typechecking makes progress.

Proof. Any backtracking in the second phase, caused by a change in the inferred type of a free identifier, is immediately followed by re-unification of all the obsolete type constraints (nothing is lost). On the other hand, a new arc is added to the DAG (something is gained). □

3.3 Remembering all Earlier Unifications (Algorithm B)

The proof of *Theorem 2* (the first phase of typechecking need never be repeated) actually showed a stronger result: no unifications of type constraints done before the unification of the obsolete type constraint need be repeated. This applies equally well to earlier unifications in the *second* phase. If these are not repeated, fewer substitutions are made obsolete; it is less likely that the type of the parent definition will change; so the paths of re-typechecking through the DAG may be shorter. These improvements are taken into account in Algorithm B, which is like Algorithm A except that the final **when** clause is replaced by the following.

```
when the inferred type of one of def's free identifiers, m, changes,
    reduce the set of substitutions to that which
    was in effect before (m) was last unified;
    flag (m) and all constraints unified since
    the last unification of (m) as not unified.
```

Figure 4: Algorithm B (differences from A)

The action of reducing the set of substitutions is similar to that which occurs when Nikhil's typechecker recovers from a type error. In DAG terms, Algorithm B demands re-typechecking of fewer of the arcs leading from the paths towards the root than Algorithm A, and these paths may be shorter.

Example (continued)

Returning to the example in which the second phase found definitions for `g`, `f`, and `h` in that order, suppose I_f changes to `num -> bool`. Algorithm B retains the substitution `{num for @d}` resulting from the unification of `(g)`, and causes only the unifications of `(f)` and `(h)` to be repeated.

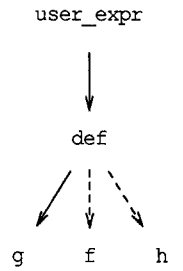


Figure 5: DAG of example for Algorithm B

E_f is still `num -> num`, which does not unify with the new I_f, `num -> bool`, and so a type error is detected in the use of `f`. The type constraint `(h)` can still be unified.

3.4 Remembering all Unaffected Unifications (Algorithm C)

When the type of a free identifier's definition changes, the obsolete type constraints of its parent definition must be determined and the substitutions resulting from their unification removed. All the typechecking algorithms discussed so far are extravagant in deciding which type constraints are obsolete: all of them (Nikhil's algorithm); all those corresponding to free identifiers (Algorithm A); and all those unified since that of the free identifier whose inferred type has changed (Algorithm B). None of these algorithms considers the composition of the constraints themselves. A finer discrimination between type constraints can be made by noting which type variables are involved in each. This is the basis of Algorithm C. Two more items of terminology will be useful.

Let V_i, where i denotes a free identifier, be the set of type variables appearing in E_i after pruning at the end of the first phase of typechecking.

Let S_i, where i denotes a free identifier, be the set of type variables for which substitutions were introduced by the most recent unification of type constraint (i).

Algorithm C is like Algorithms A and B, but the final **when** clause is replaced by the following two **when** clauses.

```
when the inferred type of one of def's free identifiers, m, changes,
    deem obsolete the type constraint (m);
when a type constraint (n) is deemed obsolete,
    remove the substitutions for variables in S_n,
    and flag (n) as not unified;
    for each unified type constraint, (p),
        if S_n ∩ V_p ≠ {},
            deem obsolete the type constraint (p).
```

Figure 6: Algorithm C (some differences from A)

Theorem 6. The process of determining obsolete type constraints terminates.

Proof. Every call flags one free identifier as not typechecked, and recurses only on typechecked free identifiers. Since there are initially only a finite number of typechecked free identifiers, termination of the process is guaranteed by induction. □

The type of a definition may change when the type constraint of one of its free identifiers is deemed obsolete. This may be detected efficiently by having another V_i-like value, V_{def}, giving the type variables used in the definition's expected type, E_{def}, after pruning at the end of the first phase. The second of the new **when** clauses is extended with the following step.

```
if S_n ∩ V_def ≠ {},
    deem obsolete all unified type constraints
    corresponding to uses of def.
```

Figure 7: Algorithm C (more differences from A)

The type of a definition may also change when the type constraint of one of its free identifiers is unified. This cannot be detected by such an efficient means. Such refinements must be detected by explicit comparison of before and after types.

Given explicit checks for refinements of the type of the parent definition, the first of the new **when** clauses is redundant.

Example (continued)

Returning to the example again, the dependence of expected types on type variables is as follows.

E_f is num -> @d so V_f = {@d}
E_g is @d -> num so V_g = {@d}
E_h is num so V_h = {}

After all free identifiers have been typechecked (in the order g, f, h) substitutions will have been computed only as a result of unifying (g).

$$S_f = \{\}$$
$$S_g = \{@d\}$$
$$S_h = \{\}$$

When I_f changes, there are no substitutions in S_f to be removed; (f) is simply flagged as not unified. A search is then made for dependent type constraints.

$$S_f \cap V_g = \{\} \cap \{@d\} = \{\}$$
$$S_f \cap V_h = \{\} \cap \{\} = \{\}$$
$$S_f \cap V_{def} = \{\} \cap \{\} = \{\}$$

It is clear from S_f being {} that no unifications of other type constraints depended on obsolete substitutions. The type constraint (h), which was deemed obsolete by both Algorithms A and B, is in fact independent of changes in I_f.

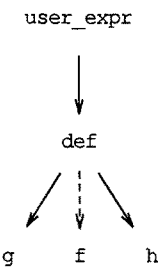


Figure 8: DAG of example for Algorithm C

3.5 Improving the Data Structures (Algorithm D)

The V_i values depend only on the results of phase one, so they are constants when used in phase two. The efficiency of an implementation may be improved by *memoing* the V_i values. Better still, they can be evaluated eagerly and represented in an inverted form as a *mapping* from type variables to the type constraints whose expected types depend on them, as follows.

$$\{\ v \rightarrow \{\ (i)\ \mid\ v \in V_i\ \}\ \}$$

In the case of the example, the collection of V_i values can be represented by the following mapping.

$$\{\ @d \rightarrow \{\ (f),\ (g)\ \}\ \}$$

When the type constraint of a free identifier is deemed obsolete, the type variables in S_j can now be mapped directly onto sets of type constraints. Those constraints that have been unified are deemed obsolete. The

V_{def} value may be represented by a similar mapping. In fact, each definition can have its own separate mappings for both its V_i values and its V_{def} value.

Algorithm D, including explicit checks for refinements of I_{def}, is given in full in Figure 9.

```
idcheck i def ->
    unify E_i with I_i, extending def's substitution;
    flag (i) as having been unified;
    if I_def is refined,
        deem obsolete all unified type constraints
        corresponding to uses of def.


typecheck def ->
  -- phase one
    Nikhil_typecheck def in a type environment
    initially containing only the primitives;
    prune I_def;
    for each free identifier, i, in def,
        prune E_i;
        flag (i) as not unified;
    compute the mapping from type vars to type constraints;
    empty def's substitution;
  -- phase two
    when the value of one of def's free identifiers, j, whose type
    constraint (j) has not been unified, is needed by a computation,
        idcheck j def;
    when a type constraint (k) is deemed obsolete,
        remove the substitutions for variables in S_k,
        and flag (k) as not unified;
        deem obsolete all unified type constraints found
        in the results of mapping over variables in S_k;
        if S_k ∩ V_def ≠ {},
            deem obsolete all unified type constraints
            corresponding to uses of def.
```

Figure 9: Algorithm D

4 Typechecking Recursive Definitions (Algorithm E)

Mutually recursive definitions must be typechecked using a single set of substitutions, since the same type variables can be involved in typechecking more than one of them. Similarly, typechecking any free identifier may refine not only the inferred type of its parent definition but also the inferred type of any definition mutually recursive with its parent definition. In this section, Algorithm D will be modified accordingly.

It is still useful to view the definitions involved in a computation and the calls between them as a DAG, but now nodes correspond to sets of mutually recursive definitions rather than individual definitions. In other words, the *condensation* of the dependency graph is a DAG. Each node in such a DAG has a single set of substitutions and a single mapping from type variables to free identifiers whose expected types involve those type variables.

4.1 Detecting Cycles

Under the automatic lazy loading mechanism, cycle detection has to be done on-the-fly. Definitions are treated as non-recursive until *every* free identifier in a cycle has been typechecked. Then *all* free identifiers in the cycle must be re-typechecked, now regarding some type variables as non-generic. In DAG terms, when a *new* arc is introduced that links a set of nodes into a new cycle, those nodes are collapsed into a single node. These operations are performed as follows, at the end of `idcheck`.

```
if i completes a new cycle among nodes,
    deem obsolete all unified type constraints
    corresponding to arcs in that cycle;
    collapse the nodes in the cycle into a single
    node, combining substitutions and mappings.
```

Figure 10: Algorithm E (some differences from D)

The obsolete type constraints must be re-typechecked before permitting the computation to proceed. The check for a new cycle involves inspecting all paths through the graph starting from the destination of the proposed new arc to see if the source of the new arc is reachable. An efficient formulation of this search has been extracted from Dijkstra's strongly connected components algorithm (Dijkstra 1976), though it can still be the major cost in phase two.

4.2 Re-typechecking

As well as checking for refinements in the type of the parent definition when a type constraint is unified or deemed obsolete, similar checks must be applied to all definitions that are mutually recursive with the parent. So relative to Algorithm D, each such check is enclosed in a loop of the following form, with the check modified to apply to definition `d`. It is intended that the range of `d` will include `def`.

```
for each definition, d, which is mutually recursive with def,
    ...d...
```

Figure 11: Algorithm E (more differences from D)

The substitution-based method first introduced in Algorithm C is still applicable in the context of mutually recursive definitions. Consequently, the extent of re-typechecking need not encompass entire cycles; this is an improvement on Nikhil's algorithm, which re-typechecks all mutually recursive definitions when any one of them needs to be re-typechecked. However, in cases where it is necessary to re-typecheck all the way round a cycle, the re-typechecking process of Figure 2 would fail to terminate: each definition in a complete cycle is the parent of at least one free identifier, so re-typechecking of a complete cycle would fail to start. The re-typechecking process must be re-formulated in terms of nodes rather than definitions.

```
while there remain free identifiers to be re-typechecked,
    let n be a node chosen such that n is not the source of any arcs
    to be re-typechecked and (n is either the destination of at least one
    such arc or n contains a cycle of definitions with free identifiers
    to be re-typechecked);
    for each free identifier, j, in a cycle within n
    that remains to be re-typechecked,
        let p be j's parent definition;
        idcheck j p True;
    for each free identifier, k, that is an arc to n
    that remains to be re-typechecked,
        let p be k's parent definition;
        idcheck k p True.
```

Figure 12: Re-typechecking (Algorithms E–F)

The timing of the re-typechecking of free identifiers in cycles is arranged to satisfy the requirement that all free identifiers in a node are re-typechecked before arcs to that node. The extra argument in the calls to idcheck is a Boolean that indicates re-typechecking in progress. If it is True, no additional re-typechecking is initiated in the same cycle; this is necessary to ensure termination.

```
idcheck i def retypecheckingp ->
    unify E_i with I_i, extending def's substitution;
    flag (i) as having been unified;
    for each definition, d, which is mutually recursive with def,
        if I_d is refined,
            for each unified type constraint j
            corresponding to a use of d,
                if not (retypecheckingp and j's parent definition
                is mutually recursive with i's definition),
                    deem obsolete type constraint j;
    if i completes a new cycle among nodes,
        deem obsolete all unified type constraints
        corresponding to arcs in that cycle;
        collapse the nodes in the cycle into a single
        node, combining substitutions and mappings.
```

Figure 13: Part of Algorithm E

4.3 Revising the DAG when Definitions are Removed

Removing a definition breaks any cycles of mutual recursion involving that definition. In DAG terms, when a definition in a collapsed node is removed, a new piece of DAG must be built from the remaining contents of the collapsed node. There may be mutually recursive cycles within the remaining definitions. These remaining cycles must be detected so that the sets of substitutions and mappings can be partitioned appropriately. The arcs between the new nodes must be re-typechecked, due to non-generic type variables becoming generic.

4.4 Correctness

All the theorems proved in Section 3.2 still hold, though further discussion of *Lemma 1* and *Theorem 3* is appropriate.

The proof of *Lemma 1* (the type constraints are correct) holds only after the mutual recursion has been detected. Prior to this, some type variables were treated as generic when they should have been non-generic; some type errors may have been overlooked, but none will have been found where none exist.

Theorem 3 (the re-typechecking process terminates) still holds if the further re-typechecking higher up the DAG really is strictly higher up the DAG and not further round a cycle. The extra argument to `idcheck` avoids such further re-typechecking around a cycle. Omitting such re-typechecking does not jeopardise correctness, as all mutually recursive definitions share the same set of substitutions and so any refinement to one of them implicitly propagates to the others.

5 Typechecking Local Definitions (Algorithm F)

Some languages do not have local definitions (Turner 1982), in which case the typechecker described above would be satisfactory. However, the consensus of language designers – including Turner (Turner 1986) – is to provide local definitions. These require further modification of the typechecking algorithm. Some more terminology will be useful.

A *locally-bound* identifier is a use of a local definition. Those identifiers previously known as *bound* identifiers are henceforth known as *parameter-bound* identifiers. These are further sub-divided into *local* parameter-bound identifiers and *non-local* parameter-bound identifiers according to whether the binding occurrence of the parameter is inside or outside the parent definition.

There is no reason to delay unification of type constraints corresponding to locally-bound identifiers until those identifiers are used in a computation. Unlike free identifiers, the definitions of locally-bound identifiers are already known. These type constraints are delayed until phase two only to reduce the cost of re-typechecking. It is therefore appropriate for them to be the first to be unified in phase two, before those of any free identifiers.

5.1 Non-Local Parameter-Bound Identifiers

Treating local definitions like top-level definitions does not work if the local definition uses any non-local parameter-bound identifiers. The problems that arise in this context are illustrated by the following example.

```
Define he x y ->
    ( let res ->
        ( let hx -> head x in hx = y )
    in res )
```

An immediate consequence of non-local parameter-bound identifiers is that a single set of substitutions must be shared by the definitions involved (since otherwise the parameters' types would not be constrained by all of their uses). Phase one infers the following types for the definitions in the example.

$$I_{he} \text{ is } @a \text{ -> } @b \text{ -> } @c$$
$$I_{res} \text{ is bool}$$
$$I_{hx} \text{ is } @d$$

Those type constraints corresponding to free and locally-bound identifiers remain to be unified.

```
@c = bool                          (res)
@b = @e                            (hx)
@a -> @d = [@f] -> @f              (head)
```

The final type that should be computed for he is `[@f] -> @f -> bool`.

Eager typechecking of locally-bound identifiers unifies both `(res)` and `(hx)`. Unifying `(res)` refines its parent's type I_{he} to `@a -> @b -> bool`. Unifying `(hx)` reveals no new information.

The problems arise when `(head)` is unified. Substitutions `[@f] for @a` and `@f for @d` are introduced. So the non-genericity of `@a` is acquired by `@d` and `@f`. But I_{hx} originated as a fresh copy of `@d`. Now that `@d` is non-generic, I_{hx} is `@d` itself. Although I_{hx} has become no less polymorphic, the change in genericity necessitates re-typechecking of all uses of hx, otherwise the constraint between `@a` and `@b` would be overlooked. The conclusion is that the present treatment of dependency, based on substitutions alone, is not adequate to propagate the refinement arising from the unification of `(head)` to the uses of hx.

The substitutions introduced by unifying `(head)` have a second effect: they constrain the type of the non-local parameter x from `@a` to `[@f]`. So all uses of he must be re-typechecked. This dependence is direct from the definition of hx to that of he: it cannot propagate indirectly through `res` because I_{res} cannot be refined in any way from its phase one value of `bool`.

Changes in Genericity

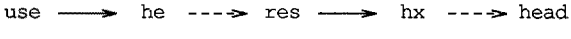
The genericity of a type variable in a type constraint depends on the context in which the atomic expression corresponding to the type constraint appears, that is whether the type variable corresponds to a parameter in scope. Fortunately, the scope rules of the language ensure that all uses of a local definition must lie within the scope of all those non-local parameters that are in the scope of the definition itself. So all changes in genericity can be detected by checking only for those in the context of the definition. The genericity change check need be done only if there is no refinement to a less polymorphic type.

Non-Local Parameter Refinements

Non-local parameter refinements can be detected by comparison of the before and after types of the function definitions that introduce those parameters. This is similar to the checks for refinements in the types of the parent definition and definitions mutually recursive with the parent. It is sufficient to check all those definitions that share the same set of substitutions. However, in the context of non-local parameter refinements it is necessary to check only those function definitions whose parameters are used by the local

definition. Note that the functions' types must be checked, not the individual parameters' types, as otherwise constraints between type variables may be overlooked.

As a consequence of non-local parameter refinements, the initial DAG of arcs to be re-typechecked may no longer be connected. This is the case for the above example, as illustrated below.



In such cases, arcs from the local definition should be re-typechecked before arcs to the function that introduced the non-local parameter, so that the parameter refinement takes place before uses of the function are re-typechecked. Otherwise, the re-typechecking of uses of the function (and any other re-typechecking higher up the DAG done before re-typechecking of arcs from the local definition) will be repeated when the non-local parameter refinement is detected during re-typechecking. This makes it even more attractive to determine the order in which to re-typecheck arcs from depths rather than from local connectivity.

Algorithm F is given in full in Figure 14. Note that it is necessary to check for genericity changes and non-local parameter refinements both on unifying a type constraint and on deeming a type constraint obsolete.

```
-- deem obsolete uses
deemobsuses def ->
    for each unified type constraint m
    corresponding to a use of def,
        deem obsolete type constraint m.

-- deem obsolete non-recursive uses
deemobsnruses i def retypecheckingp ->
    for each unified type constraint j
    corresponding to a use of def,
        if not (retypecheckingp and j's parent definition
        is mutually recursive with the definition of i),
            deem obsolete type constraint j.

idcheck i def retypecheckingp ->
    unify E_i with I_i, extending def's substitution;
    flag (i) as having been unified;
    for each definition, d, which is mutually recursive with def,
        if I_d is refined or any type variables
        in I_d have changed genericity,
            deemobsnruses i d retypecheckingp;
        for each definition, e, that introduces a
        non-local parameter-bound identifier in d,
            if I_e is refined,
                deemobsnruses i e retypecheckingp;
    if i completes a new cycle among nodes,
        deem obsolete all unified type constraints
        corresponding to arcs in that cycle;
        collapse the nodes in the cycle into a single
        node, combining substitutions and mappings.
```

```
typecheck def ->
  -- phase one
    Nikhil_typecheck def in a type environment initially containing only
    the primitives and extended only with parameter-bound identifiers;
    prune I_def;
    for each free or locally-bound identifier, i, in def,
        prune E_i;
        flag (i) as not unified;
    compute the mapping from type vars to type constraints;
    empty def's substitution;
  -- phase two
    for each locally-bound identifier, i, in def,
        idcheck i def False;
    when the value of one of def's identifiers, j, whose type
    constraint (j) has not been unified, is needed by a computation,
        idcheck j def False;
    when a type constraint (k) is deemed obsolete,
        remove the substitutions for variables in S_k,
        and flag (k) as not unified;
        deem obsolete all unified type constraints found
        in the results of mapping over variables in S_k;
        for each definition, d, which is mutually recursive with def,
            if S_k ∩ V_d ≠ {} or any type variables
            in I_d have changed genericity,
                deemobsuses d;
            for each definition, e, that introduces a
            non-local parameter-bound identifier in d,
                if I_e is refined,
                    deemobsuses e.
```

Figure 14: Algorithm F

6 Type Declarations and Type Errors

If a declared type D_i is given for a free identifier i, the type constraint $E_i = I_i$ is split into two type constraints: $E_i = D_i$ and $D_i = I_i$. Only I_i can change, and so the type constraint $E_i = D_i$ may be unified in the first phase. The type constraint $D_i = I_i$ is the one denoted by (i) in the second phase. To be more precise, (i) should be $D_i \leq I_i$ since a definition's type must be at least as polymorphic as its declared type. (Actually, I_i is not the only type expression that can change: explicitly-defined types can be redefined, say from an algebraic type to a synonym type, so requiring all type constraints involving D_i to be re-unified. In this circumstance, glide repeats the whole of the first phase.)

The new typecheckers can give similar diagnostics of type errors as Nikhil's algorithm. However, they are more likely to attribute a type error to a free identifier, since constraints due to free identifiers are unified last. Consider the case of a function application in which the function is a free identifier, for example `transpose 42` where the type of `transpose` is `[[@a]] -> [[@a]]`. The diagnostic generated is as follows.

```
Type error in: transpose
Expected type: num -> @b
Inferred type: [[@a]] -> [[@a]]
```

Or to paraphrase,

```
transpose cannot be applied to 42.
```

Treating the function as the type erroneous component in this application can be justified by stating that the types of literals are clear, whereas the definitions of free identifiers are given elsewhere and so should be treated with greater suspicion. Nikhil takes a different viewpoint (Nikhil 1985).

> "Our algorithm favours the function part by typechecking it first and allowing it to determine the expected type of the argument part. This is based on the observation that in most cases the function part is not a complicated expression and is less likely to be the cause of the type error in the programmer's judgement."

Hence a diagnostic is given about the argument value being type erroneous.

```
Type error in: 42
Expected type: [[@a]]
Inferred type: num
```

Or to paraphrase,

```
42 is an inappropriate argument for transpose.
```

Of course, neither diagnostic is guaranteed to be appropriate. The point is that, for the given example, our typechecker has no choice but to give the former diagnostic, whereas Nikhil's could give either and he happens to recommend the latter.

7 Cost

The table in Figure 15 compares the performance of Nikhil's algorithm with that of Algorithm F when loading two programs. For each case, the performance is given by a triple of measurements. The first is the number of unifications of type constraints performed. The second is the number of *definitions* re-typechecked (in the case of Nikhil's algorithm), or the number of *free identifiers* re-typechecked (in the case of Algorithm F). The third is the effort expended in re-typechecking as a proportion of the cost of typechecking the loaded definitions alone (measured as number of unifications). The measurements for Algorithm F are for complete typechecking of the programs, not just those bindings required by a particular computation. The programs are omitted here due to lack of space, but are reproduced in full elsewhere (Toyn 1987).

The number of unifications done by Nikhil's algorithm under eager loading should equal the number of unifications excluding those in re-typechecking done by Algorithm F under lazy loading. This is true for the `queens` program (`199 - 19 = 180`); the small discrepancy for the `edit` program (`384 - 47 > 334`) is due to one of the auxiliary definitions needed for Algorithm F being primitive in the version of glide with Nikhil's algorithm.

Program	#Defines	Nikhil eager	Nikhil lazy	Alg. F lazy
queens n	10	180, 0, 0%	408, 16, 127%	199, 19, 10%
edit	14/15	334, 0, 0%	749, 18, 124%	384, 47, 14%

Figure 15: Comparison of costs for loading

The figures show that Algorithm F is an order of magnitude more efficient than Nikhil's algorithm at re-typechecking in the context of definitions being loaded lazily; the total typechecking cost is halved. Also, for Algorithm F, the number of type constraints unified in phase one as a proportion of all type constraints was 90% for the queens program and 86% for the edit program. These figures verify the original hypothesis that most of the type constraints remain the same, indeed these type constraints can never change.

The other context in which re-typechecking is required is when a definition is modified. Algorithm F performs some re-typechecking when the old version of a modified definition is removed, and more re-typechecking when the new version is introduced. In contrast, Nikhil's algorithm performs re-typechecking based on the difference between the types of the old and new versions. Such an optimization remains to be incorporated into Algorithm F. An example arose with the edit program: when the editor was changed to support an *undo* command, a parameter that previously represented the text became a list of past texts. This change involved modifications to several definitions, the costs of which are tabulated in Figure 16. The pairs of measurements are total unification counts and either definition or unification counts during re-typechecking.

Define	Nikhil	Alg. F
display	46, 1	34, 8
do	82, 1	77, 13
decode	58, 0	68, 23
edit	16, 0	18, 11

Figure 16: Comparison of costs for modifying

These figures suggest that Algorithm F and Nikhil's algorithm have similar performance. However, note that Algorithm F's re-typechecking algorithm is non-optimal and also the optimization mentioned above.

8 Discussion

An alternative approach to our problem with glide would have been to adjust the loading sequence to avoid re-typechecking. But it is impossible to avoid all re-typechecking if there are mutually recursive definitions. Furthermore, lazy loading avoids the cost of typechecking unused definitions.

Another consequence of the lazy loading strategy is that a type error may not be found until a computation has already made successful use of the identifier's definition. This may seem undesirable, but is a best compromise: the exploratory style which lazy loading supports is far more valuable than the detection of all type errors before execution. If a type error is discovered for such an identifier during re-typechecking, the computation is aborted. When the programmer can think of no more test computations and is stuck for which definition to modify next, an explicit request can be made to the system to search for type errors. Until that time, the programmer is free to develop the program without being inconvenienced. Certainly such a search should be requested before making any claim that the program works.

9 Conclusions and Further Work

In the context of an exploratory programming environment, it is a requirement that re-typechecking be performed efficiently. This requirement is difficult to satisfy in the context of lazy loading, such as results from use of a persistence mechanism. This paper has presented incremental polymorphic typechecking algorithms that are suitably efficient. The performance of the new algorithms is not seriously affected by the loading strategy used. Consequently they are significantly better in the context of lazy loading, and not significantly worse in the context of eager loading.

A linear re-typechecking algorithm would be preferred, that is one that re-typechecks an arc only when there are no lower arcs waiting to be re-typechecked. Computing explicit depth information could be expensive: whenever an arc is added to the DAG, the depths of all nodes accessible via the new arc may have to be revised. An alternative would be for each node to have a link to its deepest parent. This would decrease the cost of maintaining the depth information, but increase the cost of using it. Another alternative is to thread all nodes of the complete graph in a sequence consistent with either the higher or lower partial orders, then the re-typechecking process could do a single sweep of the nodes in this sequence. Maintaining the sequence could be done as a side effect of mutual recursion detection. However, both maintaining and using the sequence would have a cost linear in the number of nodes in the complete DAG. This may not be significantly better than the simple re-typechecking algorithm's quadratic cost in the number of nodes in the DAG of arcs to be re-typechecked.

The cost comparison of the two typecheckers assumed that the cost of a unification is the same in both typecheckers. This was true for the given implementations. However, more efficient unification algorithms exist that may not be applicable in the context of the new typechecker. Examination of Martelli and Montanari's *An Efficient Unification Algorithm* (Martelli 1982) suggests that it would be applicable only to the first phase of Algorithm F. This is because Martelli and Montanari's algorithm assumes that it is free to determine in which order to unify the type constraints, whereas phase two of Algorithm F imposes the

sequence in which the type constraints are to be unified. On the other hand, the separation of the set of substitutions into localised subsets, each emptied between phases, means that the inefficient unification algorithm may not be so bad. A cost comparison of the two typecheckers in the context of an efficient unification algorithm would be interesting.

Acknowledgement

This work was funded by the UK Science and Engineering Research Council.

References

Cardelli 1985. L. Cardelli, "Basic Polymorphic Typechecking", *Polymorphism* **II**(1) (January 1985).

Dijkstra 1976. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. (1976).

Martelli 1982. A. Martelli and U. Montanari, "An Efficient Unification Algorithm", *ACM TOPLAS* **4**(2), pp. 258-282 (April 1982).

Milner 1978. R. Milner, "A Theory of Type Polymorphism in Programming", *J. of Computer and System Sciences* **17**(3), pp. 348-375 (December 1978).

Nikhil 1985. R. S. Nikhil, "Practical Polymorphism", pp. 319-333 in *Functional Programming Languages and Computer Architecture*, ed. Jean-Pierre Jouannaud, Springer-Verlag Lecture Notes in Computer Science 201 (September 1985).

Robinson 1965. J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", *JACM* **12**(1), pp. 23-41 (January 1965).

Toyn 1986. I. Toyn and C. Runciman, "Adapting Combinator and SECD Machines to Display Snapshots of Functional Computations", *New Generation Computing* **4**(4), pp. 339-363 (December 1986).

Toyn 1987. I. Toyn, "Exploratory Environments for Functional Programming", DPhil thesis (unpublished), Dept. of Computer Science, University of York (April 1987).

Turner 1982. D. A. Turner, "Recursion Equations as a Programming Language", in *Functional Programming and its Applications*, ed. J. Darlington, P. Henderson, and D. A. Turner, Cambridge University Press (1982).

Turner 1985. D. A. Turner, "Miranda: A Non-Strict Functional Language with Polymorphic Types", pp. 1-16 in *Functional Programming Languages and Computer Architecture*, ed. Jean-Pierre Jouannaud, Springer-Verlag Lecture Notes in Computer Science 201 (September 1985).

Turner 1986. D. A. Turner, "An Overview of Miranda", *SIGPLAN Notices* **21**(12), pp. 158-166 (December 1986).