INCREMENTAL POLYMORPHIC TYPE CHECKING IN *B*

Lambert Meertens†

Mathematical Centre
P.O.B. 4079, 1009 AB Amsterdam, The Netherlands

Abstract

The programming language *B* has been designed for personal computing. In *B*, variables need not be declared, nor formal parameters specified. Nevertheless, *B* is strongly typed. All type requirements can be checked statically. To signal type violations on the spot during editing, the computations can be organized so that local modifications to the source text require a modest amount of recomputation.

0. INTRODUCTION

The programming language *B* (MEERTENS[9]) has been designed for personal computing. The aim has been to provide an alternative to popular languages such as BASIC, which combine simplicity with a deplorable lack of support for structured-programming methods. One of the design requirements for *B* was that it had to be strongly typed, including the possibility of checking the type requirements completely statically. On the other hand, the freedom from the duty of declaring variables is certainly an attractive property for a language to be used for personal computing. The power of type-finding algorithms for weakly typed languages (see, e.g., TENENBAUM[13]) if the programs adhere to a self-imposed type discipline led to an investigation into the possibility of combining the two desiderata. Informal arguments convinced us of the feasibility. So we designed *B* so that variables need not be declared, nor formal parameters specified. Nevertheless, it was proclaimed that *B* is strongly typed, and that all type requirements could be checked statically. Only afterwards, when we started to worry about an algorithm for the type check, did we become aware that essentially the same idea, known as "type polymorphism", had already been implemented successfully (MILNER[10]).

† Address until September 1983: Courant Institute of Mathematical Sciences, New York, New York.

Another design criterion for *B* was that it should fit snugly into a *B*-dedicated environment. As in APL, *B* "units", as procedure definitions in *B* are called, exist in a workspace. Units can be invoked from other units, but also directly by the user. So *B* itself can be used as the command language for such an environment, and instead of "files", ordinary *B* values kept in the workspace will do. An important part of this environment is the editor for composing *B* units. The editor should know enough about *B* itself to assist the user. It is desirable that the editor can warn the user on the spot about type violations. This is only feasible if the computations involved in type checking can be organized "incrementally", i.e., so that a small modification requires a small amount of (re)computation.

This paper shows how this may be done. Sections 1 to 4 may also be read as an informal treatment of type polymorphism that is easier to follow for language designers and implementers than the rather formal treatment of MILNER[10]. The organization of the paper is as follows. Section 1 gives just enough information about *B* to make the examples in the following sections understandable. Section 2 discusses the semantics of types and type requirements. The necessary theory is similar in content to that given by MILNER[10], but the approach is rather different. Section 3 relates the Unification Algorithm of ROBINSON[11] to type checking. Section 4 gives the non-incremental algorithms for computing the type requirements for a set of units. Section 5 shows how to organize the computations for incremental type checking. Section 6 deals with some issues specific to polymorphic type checking in *B*. Finally, the relationship to other work and issues for further study are discussed in section 7.

1. TYPES AND UNITS IN *B*

1.1. The types of *B*

Much of the ease of use of *B* is due to its type system.

The basic types are numbers and texts. Several values may be grouped in a tuple to form a "compound" value. The type of such a compound is determined by the types of its fields. If a unit contains a command

```
PUT 1, 'one', 'een' IN ved
```

(the PUT command is the assignment in *B*), then the variable `ved` must have the type "compound-with-numeric-textual-textual-fields". An attempt to put a value of a different type

in ved, as in

```
PUT 2, 1+1, 'twee' IN ved
```

constitutes a type violation.

"Lists" are another way of grouping values, but these values must all have the same type, and this determines the type of the list. In fact, a list is a bag (multiset).

"Tables" are generalized arrays (maps): under "keys" (indexes) of any type, "associates" (again of any type) may be stored. For a given table, all keys must have the same type, as must its associates (though not necessarily the same type as the keys).

1.2. Units

Units define a command or a function. They correspond to what are usually called "procedure definitions". The following unit introduces a new, user-defined SWAP command to swap the contents of two variables:

```
HOW'TO SWAP a AND b:
    PUT b, a IN a, b.
```

Once defined, this command may be used to swap the contents of two numeric variables, of two textual variables, or of any two variables of the same type. It is a type violation to attempt to swap, e.g., a numeric variable with a textual variable.

Commands and functions may not be transmitted as parameters in *B*. This restriction makes it possible to obtain a complete type checking system. "Completeness" means that semantic compliance with the strong typing requirements of a set of units implies that they are also syntactically acceptable. This issue is further discussed in section 7.

Next to units defining commands, *B* has function definitions, as in

```
YIELD cosh x:
    PUT exp x IN ex
    RETURN (ex+1/ex)/2.
```

The easiest way to treat these is to observe that a function definition can be transformed to a command definition, as in

```
HOW'TO COSH x GIVING y:
    PUT exp x IN ex
    PUT (ex+1/ex)/2 IN y.
```

For all formulae involving the function, a new temporary variable is created taking its place, and a call of the command is inserted with that temporary variable as the result parameter. This makes it possible to explain the type checking of *B* in terms of command units only.

B also allows "parameter passing" through global variables. These variables have to be listed at the start of the unit, following the keyword SHARE, as in

```
HOW'TO PUSH v:
    SHARE stack
    PUT v IN stack[#stack+1].
```

An easy way to treat these is to make the parameter explicit, as in

```
HOW'TO PUSH v ON stack:
    PUT v IN stack[#stack+1].
```

All calls of PUSH have to be amended to incorporate the extra parameter. This will, in general, necessitate the introduction of extra parameters in other units invoking PUSH directly or indirectly.

Finally, *B* units may contain "refinements", as in

```
YIELD cosh x:
    RETURN (ex+1/ex)/2
ex: RETURN exp x.
```

Here, ex is an expression whose meaning is defined by a refinement, given as the last line of the unit. Command refinements require no special measures. Expression refinements can be handled similarly to function definitions.

2. TYPES AND TYPE REQUIREMENTS

2.1. The structure of types

To make the theory of type polymorphism apply to a language, it must be possible to express its types as built from a set of formation rules, each of the form

$$\text{type: } <\text{atom}, \text{type}_1, \ldots, \text{type}_n>, \quad n \geq 0.$$

For a given atom, there is one formation rule. The types are formed by applying these rules a finite number of times, starting from the basic types of the form <atom>. The rules may be interpreted as constituting a context-free tree grammar, and the types are then trees whose nodes are labeled with atoms. The leaf nodes are basic types.

For *B*, we have the rules

type: <NUMBER>;
<TEXT>;
<COMPOUND2, type_1, type_2>;
<COMPOUND3, type_1, type_2, type_3>;
⋮
<COMPOUNDi, type_1, type_2, ... , type_i>;
⋮
<LIST, type_1>;
<TABLE, type_1, type_2>.

An example of a type that may be built from these rules is

<TABLE, <TEXT>, <LIST, <NUMBER>>>.

This would be the type of a table, indexed by texts, under which lists of numbers are stored.

2.2. Type requirements

Suppose that we have a *B* unit

```
HOW'TO SHOW t AT k:
    WRITE t[k].
```

From the expression t[k] we see that t is a table, and that a selection on that table is performed, using as selection key the value of k. This gives information about the type of t (some table) and a relation between the types of t and of k (the type of the keys of t must be that of k). If we denote the type of t by τ and that of k by κ, this information may be expressed in one logical formula:

$$\exists \alpha_1, \alpha_2 \colon \tau = <\text{TABLE}, \alpha_1, \alpha_2> \ \& \ \kappa = \alpha_1.$$

The bound variables α_1 and α_2 range, of course, over all types. The formula may be expressed still more conveniently (in view of later developments) as

$$\exists \alpha \colon (\tau, \kappa) = (<\text{TABLE}, \alpha_1, \alpha_2>, \alpha_1).$$

α ranges over all pairs of types, and, in the general case, over all tuples of types for all components α_i occurring in the formula. This formula contains all information we can hope to

glean from the unit about the types of its formal parameters. It can be interpreted, for the time being, as a condition that must be satisfied at run time: the types of any actual parameters provided (at run time) for `t` and `k` must then fit the given pattern for some types α_1 and α_2. Note that this poses a requirement not only on `t` and `k` *individually*, but also on the *combination*. For example, the following two combinations are both acceptable as types for the actual parameters to a `SHOW` command:

(*i*) $(\tau_a, \kappa_a) = (<\text{TABLE}, <\text{NUMBER}>, <\text{TEXT}>>, <\text{NUMBER}>)$;

(*ii*) $(\tau_a, \kappa_a) = (<\text{TABLE}, <\text{TEXT}>, <\text{NUMBER}>>, <\text{TEXT}>)$;

but this third one does not fit the pattern, even though it is made up from pieces taken from the acceptable combinations (*i*) and (*ii*):

(*iii*) $(\tau_a, \kappa_a) = (<\text{TABLE}, <\text{NUMBER}>, <\text{TEXT}>>, <\text{TEXT}>)$.

Now consider another unit, invoking the `SHOW` command whose type requirement we have just analyzed:

```
HOW'TO PRODUCE n FROM s:
    FOR i IN {1..n}: SHOW s AT i.
```

The expression `{1..n}` informs us that ν, the type of `n`, must be that of `1`, which is manifestly <NUMBER>. Now, if the value of `n` (at run time) is not positive, the list of values through which `i` must run is empty, and no value is put in the variable `i`, nor will the `SHOW` command be executed. It might appear that (since we do not know the value of `n` in advance) we cannot impose restrictions on `s`. This would be true if we were just doing an analysis for a language without strong typing, to see if some dynamic type checks may be foregone. But because we are concerned with strong (static) type checking, our requirements are stronger. We require the type of the actual parameters for `SHOW` to be consistent (i.e., fitting the pattern) with the formal parameters, whether or not that command will be actually reached. Similarly, we require the type of `i` to be <NUMBER> (or, equivalently, to be the same as ν), even though we do not know if `i` will, dynamically, receive a value. Even if we could deduce that `i` would never receive a value—e.g., if the range expression were `{n+1..n}`—the requirement would still be the same. This is consistent with the static typing requirements in languages with declarations: in ALGOL 60, for example, the block

```
begin Boolean i;
      for i := n+1 step 1 until n do
      show (s, i)
end
```

contains a type violation.

We are now ready to proceed with the type analysis for the unit defining the `PRODUCE` command. We want to retrieve information from the body about ν and σ, the types of the formal parameters, but also about ι, the type of the local variable. The latter will no longer be of interest after the analysis is done, but is needed during the analysis, both to check the type consistency of the various uses of `i`, and because it serves as an intermediary to express local information about ν and σ. The information derived from the expression `{1..n}` may be expressed by

$$\exists\alpha\colon (\nu, \sigma, \iota) = (<\text{NUMBER}>, \alpha_1, \alpha_2).$$

This formula could be expressed much more concisely, but is written thus to achieve uniformity with the following formulae. The requirement imposed on ι by the use of `i` as running variable for the `FOR` command is

$$\exists\alpha\colon (\nu, \sigma, \iota) = (\alpha_1, \alpha_2, <\text{NUMBER}>).$$

Note that the bound variable α_1 in the previous formula does not correspond to α_1 in this one, but to α_2. In general, bound variables in one formula need not correspond to identical variables in another formula. The reason why this obvious fact is nevertheless stressed will become apparent later on.

Finally, we come to the invocation of `SHOW`. The type of the actual parameters, (σ, ι), has to fit the pattern derived earlier on for the type of the formal parameters. This gives rise to the formula

$$\exists\alpha\colon (\sigma, \iota) = (<\text{TABLE}, \alpha_1, \alpha_2>, \alpha_1),$$

obtained by substituting (σ, ι) for (τ, κ) in the formula derived for `SHOW`. This can be brought in the same format as the two formulae already found:

$$\exists\alpha\colon (\nu, \sigma, \iota) = (\alpha_1, <\text{TABLE}, \alpha_2, \alpha_3>, \alpha_2).$$

To summarize, we have now three requirements:

(*i*) $\exists\alpha\colon (\nu, \sigma, \iota) = (<\text{NUMBER}>, \alpha_1, \alpha_2)$;

(*ii*) $\exists\alpha\colon (\nu, \sigma, \iota) = (\alpha_1, \alpha_2, <\text{NUMBER}>)$;

(*iii*) $\exists\alpha\colon (\nu, \sigma, \iota) = (\alpha_1, <\text{TABLE}, \alpha_2, \alpha_3>, \alpha_2)$.

These three requirements must be satisfied simultaneously, which can be expressed by the conjunction (*i*) & (*ii*) & (*iii*). For longer units, the formula expressing the type requirements would then soon grow very fast, making polymorphic type checking impractical. Fortunately, the conjunction is again equivalent to a formula of the same form as its constituents, viz.

$$\exists\alpha\colon (\nu, \sigma, \iota) = (<\text{NUMBER}>, <\text{TABLE}, <\text{NUMBER}>, \alpha_1>, <\text{NUMBER}>).$$

As we shall see, it is not a coincidence that the conjunction may be so expressed.

Now the analysis is done, we lose our interest in ι: the unit defining `PRODUCE` is by itself correct, but might be invoked incorrectly. This depends only on the types of the actual parameters substituted for ν and σ, so we take the "projection" of the "internal" type requirement on these two, to obtain the "external" type requirement. This yields, finally,

$$\exists\alpha\colon (\nu, \sigma) = (<\text{NUMBER}>, <\text{TABLE}, <\text{NUMBER}>, \alpha_1>).$$

It is, of course, not always possible to bring a conjunction into the desired format. Otherwise, we would never find a type violation. One example is given by

(*i*) $\exists\alpha\colon (\xi) = (<\text{NUMBER}>)$;

(*ii*) $\exists\alpha\colon (\xi) = (<\text{LIST}, \alpha_1>)$.

This is a straightforward type clash. There are more complicated situations that may arise, a simple example of which is given by

(*i*) $\exists\alpha\colon (\xi, \eta) = (<\text{LIST}, \alpha_1>, \alpha_1)$;

(*ii*) $\exists\alpha\colon (\xi, \eta) = (\alpha_1, <\text{LIST}, \alpha_1>)$.

If there were an infinite type $<\text{LIST}, <\text{LIST}, <\text{LIST}, \ldots>>>$, it would be a—in fact, the unique—type satisfying both formulae when substituted for ξ and η. However, it is understood that types are built from the formation rules by applying these a *finite* number of times, so that no types can satisfy the two formulae simultaneously.

2.3. Polytypes

For the further discussion, we must be able to talk about the kind of "type schemas" that figured in the formulae: "unfinished" type forms still containing variables.

Let the set of "polytypes" be the set of forms that can be generated from the formation rules for types, if to the given rules also are added the rules

$$\text{type: } \alpha_i, \quad i = 1, 2, \ldots .$$

In other words, we also allow type variables in polytypes. Just as with types, polytypes may be viewed as trees, but now the leaves may also be type variables. The original types, not containing type variables, will from now on be called "monotypes". (Note that monotypes are a special case of polytypes.) The Greek letter π will denote a polytype. It is expedient to identify a tuple of polytypes $(\pi_1, \ldots, \pi_n)$ with the singleton polytype $<\text{COMPOUND}n, \pi_1, \ldots, \pi_n>$. (We drop here the requirement from *B* proper that $n \geqslant 2$.) This makes it unnecessary to refer each time to tuples. The formulae found above are then all of the form

$$(*) \quad \exists \alpha: \phi = \pi,$$

where ϕ stands for the composition of the monotypes of the relevant formal parameters and local variables (to which global variables have been added as described in section 1.2).

3. UNIFICATION

3.1. Combining type requirements by unifying polytypes

Let $\text{Types}(\pi)$ stand for the set of monotypes that fit the polytype π, or, more formally,

$$\text{Types}(\pi) =_{def} \{\tau \mid \exists \alpha: \tau = \pi\}.$$

In particular, if π is a monotype, then $\text{Types}(\pi) = \{\pi\}$. On the other hand, $\text{Types}(\alpha_1)$ stands for the set of all monotypes. Note that in $\text{Types}((\alpha_1, \alpha_1))$, for example, the variable α_1 has the *appearance* of being free, but is actually bound by the $\exists \alpha$ in the definition. This is highly irregular, but convenient. The general form $(*)$ above can now be rewritten as

$$\phi \in \text{Types}(\pi).$$

As we have seen, we had to take the conjunction of all contributing formulae to obtain a formula giving the grand total of type information to be extracted from a unit. Since a conjunction

$$\phi \in \text{Types}(\pi') \ \& \ \phi \in \text{Types}(\pi'')$$

is equivalent to

$$\phi \in \text{Types}(\pi') \cap \text{Types}(\pi'')$$

we may also take the intersection of sets of monotypes. The hope is now that if that intersection is non-empty, we can find π "unifying" π' and π'', i.e., such that

$$\text{Types}(\pi) = \text{Types}(\pi') \cap \text{Types}(\pi'').$$

Such a π, if it can be found, will be denoted by $\pi' \cap \pi''$. In general, the form is not unique, because trivial renamings may be applied to the type variables without change in meaning. Such different forms will be considered equal, in the same way that $\lambda x: x+1$ and $\lambda y: y+1$ are equal. To express that two forms cannot be unified, it is convenient to introduce the "pseudo-type" $<*>$, where, by convention, $<*> \in \text{Types}(\pi)$ for any π. This convention will make it possible to define $\cap$ between polytypes in such a way that, without restrictions,

$$\text{Types}(\pi' \cap \pi'') = \text{Types}(\pi') \cap \text{Types}(\pi'').$$

Note that it follows that $\cap$ between polytypes (if we succeed in defining it) is commutative and associative.

An example of a monotype fitting a polytype is given by

$$(<\text{TABLE}, <\text{NUMBER}>, <\text{TEXT}>>, <\text{NUMBER}>) \in \text{Types}(<\text{TABLE}, \alpha_1, \alpha_2>, \alpha_1).$$

The monotype can be obtained by "refining" the polytype, by substituting $<\text{NUMBER}>$ for α_1 and $<\text{TEXT}>$ for α_2. This is precisely the meaning of the definition of Types: for an arbitrary monotype ϕ, we have $\phi \in \text{Types}(\pi)$ iff ϕ can be obtained by refining π by a suitable choice of substitutes for the type variables in π. The substitution needed in this example will be denoted by

$$[\![(<\text{NUMBER}>, <\text{TEXT}>) \rightarrow (\alpha_1, \alpha_2)]\!].$$

In the general case, we find a tuple of polytypes on the left-hand side, and a tuple (of the same length) of *distinct* type variables on the right-hand side. (It is helpful to read "replaces" for the arrow.) Note that we do not require that the substitutes are monotypes: we have use for more general substitutions. The application of a substitution is simply written by postfixing the substitution to the polytype it is applied to. If Θ stands for the above substitution (the Greek letter Θ will denote substitutions), then

$$(<\text{TABLE}, <\text{NUMBER}>, <\text{TEXT}>>, <\text{NUMBER}>) = (<\text{TABLE}, \alpha_1, \alpha_2>, \alpha_1)\Theta.$$

Using this notation, we have

$$\phi \in \text{Types}(\pi) \ \text{ iff } \ \exists \Theta: \phi = \pi\Theta.$$

So

$$\text{Types}(\pi) = \{\tau \mid \exists \Theta: \tau = \pi\Theta\}.$$

(The τ still stands, of course, for a monotype.)

We return now to the problem of finding, for given π' and π'', $\pi = \pi' \cap \pi''$. The requirement π has to satisfy can be reformulated as: for every monotype ϕ,

$$\exists \Theta: \phi = \pi\Theta \ \text{ iff } \ \exists \Theta', \Theta'': \phi = \pi'\Theta' = \pi''\Theta''.$$

It is here that the Unification Algorithm of ROBINSON [11] can be used.

3.2. The Unification Algorithm

Although the Unification Algorithm was formulated and used originally in the context of automated theorem proving, it manipulates symbolic expressions and does not depend on the interpretation assigned to them. It is described here, using the terminology already developed. To express what the Unification Algorithm achieves (not *how* it achieves this), we need some definitions.

A polytype π' is a "refinement" of a polytype π if $\pi' = \pi\Theta$ for some substitution Θ. (From the fact that the consecutive application of two substitutions Θ and Θ' may always be obtained by a single substitution Θ'', it is now immediate that $\text{Types}(\pi') \subseteq \text{Types}(\pi)$.)

A substitution Θ_u is called a "unifier" of two polytypes π' and π'' if $\pi'\Theta_u = \pi''\Theta_u$, in which case π' and π'' are said to be "unifiable", and the form obtained by applying the substitution is called a "unification" of π' and π''. It is, moreover, called a "most general" unifier, if the unification π obtained thus is such that each unification of π' and π'' is a refinement of π.

The Unification Algorithm describes how to determine, given two polytypes π' and π'', whether these are unifiable, and, if so, how to construct a most general unifier. For our purposes, the unification found is not yet exactly what we need for $\pi' \cap \pi''$, but comes very close. First, however, the Algorithm will be given. It follows the description of BOYER & MOORE[2], rather than that of ROBINSON. The Algorithm traverses the structure of the polytypes under consideration, by recursive descent.

Unification Algorithm, applied to π' and π'':
Circularity Clause:
If the Algorithm calls, during its execution, for the application of a substitution $[\![\pi \rightarrow \alpha_i]\!]$ for some type variable α_i, and π contains α_i properly as a component (so π itself is composite):
It is determined that the given polytypes are not unifiable, and the Algorithm is complete.

- If the Algorithm is at the top level (not recursively invoked):
  Let α be a tuple of all type variables occurring in π' and/or π'' and let θ be a copy of α, and put π', π'' and θ under consideration.
- Select an applicable clause:
  - π' is some type variable:
    Apply the substitution $[\![\pi'' \rightarrow \pi']\!]$ to all polytypes under consideration, replacing them by the new forms.
  - π'' is some type variable:
    Apply, as above, the substitution $[\![\pi' \rightarrow \pi'']\!]$.
  - (neither π', nor π'', is a type variable, so) π' is of the form $<\text{atom}', \pi_1', \ldots, \pi_m'>$ and π'' is of the form $<\text{atom}'', \pi_1'', \ldots, \pi_n''>$:
    - If atom' $\neq$ atom'':
      It is determined that the given polytypes are not unifiable, and the Algorithm is complete.
    - Otherwise (m and n are now known to be equal, since there is only one formation rule for a given atom):
      - Put all forms $\pi_1', \ldots, \pi_n'$ and $\pi_1'', \ldots, \pi_n''$ under consideration.
      - For $i = 1, 2, \ldots, n$:
        Apply the Unification Algorithm (recursively) to the forms (possibly modified by previous substitutions) π_i' and π_i''.
      - The (possibly modified) forms $\pi_1', \ldots, \pi_n'$ and $\pi_1'', \ldots, \pi_n''$ may now be regarded as no longer under consideration.
- If the Algorithm is (back) at the top level:
  The given polytypes have been determined to be unifiable and the substitution $[\![\theta \rightarrow \alpha]\!]$ is a most general unifier.

The Unification Algorithm will now be shown in action on a simple example. For legibility, $<\text{COMPOUND}n, \pi_1, \ldots, \pi_n>$ is abbreviated to $(\pi_1, \ldots, \pi_n)$ and $<\text{TABLE}, \pi_1, \pi_2>$ to $\pi_1 : \pi_2$. The problem is to unify $\pi' = (\alpha_1, \alpha_2 : \alpha_3)$ and $\pi'' = (\alpha_3 : \alpha_2, \alpha_1)$. The α from the description is $(\alpha_1, \alpha_2, \alpha_3)$, and θ is set to α. Since π' and π'' are composite and agree in form, the Unification Algorithm is applied recursively to the pairs of corresponding components. First, α_1 and $\alpha_3 : \alpha_2$ have to be unified. This requires the application of the substitution $[\![\alpha_3 : \alpha_2 \rightarrow \alpha_1]\!]$. This gives $\pi' = (\alpha_3 : \alpha_2, \alpha_2 : \alpha_3)$, $\pi'' = (\alpha_3 : \alpha_2, \alpha_3 : \alpha_2)$ and $\theta = (\alpha_3 : \alpha_2, \alpha_2, \alpha_3)$. Next, the second components of π' and π'' have to be unified. These have been modified by the substitution, and are now $\alpha_2 : \alpha_3$ and $\alpha_3 : \alpha_2$. Again, the forms to be unified are composite. Unification of α_2 and α_3 may be achieved by either of the substitutions $[\![\alpha_3 \rightarrow \alpha_2]\!]$ and $[\![\alpha_2 \rightarrow \alpha_3]\!]$. We select, arbitrarily, the first one, and obtain $\pi' = (\alpha_3 : \alpha_3, \alpha_3 : \alpha_3)$, $\pi'' = (\alpha_3 : \alpha_3, \alpha_3 : \alpha_3)$ and $\theta = (\alpha_3 : \alpha_3, \alpha_3, \alpha_3)$. The next forms to be unified are now, after this substitution, α_3 and α_3, which results in a dummy substitution. Climbing back from the recursion, we find no more tasks to be performed, so we have found a most general unifier for π' and π'', viz.

$$[\![(\alpha_3 : \alpha_3, \alpha_3, \alpha_3) \rightarrow (\alpha_1, \alpha_2, \alpha_3)]\!].$$

BOYER & MOORE[2] do not give a correctness proof, but a few informal remarks may make the correctness obvious. First, we remark that if the Algorithm determines unifiability, then π' and π'' are indeed unifiable; in fact, the substitutions performed have changed π' and π'' into identical forms. For this is obviously true if one of the two was originally a type variable, and from there on it follows by induction on the structure of the polytypes. The Algorithm may perform substitutions on components that had already been unified, but since these were already identical then, they will remain identical. Also, at any time during the execution of the Algorithm, the substitution $[\![\theta \rightarrow \alpha]\!]$ represents the sequence of elementary substitutions hitherto performed, so it must, at the end, be a unifier. But is it a most general unifier? To see that it is, it must be realised that if we wish to achieve unification, the elementary substitutions are obligatory, and could not have been chosen more general. It follows again by induction that the final unifier is most general. This shows already that if the Algorithm fails to produce a unifier, unification is impossible. To make this implicit argument explicit, we consider the two conditions under which non-unifiability is determined. If an atom clash occurs, unification is obviously locally impossible, and therefore globally, since the substitutions already performed were forced and quite general, and preserved unifiability if it existed. Regarding the Circularity Clause, each substitution $\Theta = [\![\pi \rightarrow \alpha_i]\!]$ prescribed is needed to unify π and α_i. But if π contains α_i as a component, then we would have $\pi\Theta \neq \alpha_i\Theta = \pi$, so unification is impossible. Finally, we can see that the Algorithm terminates, by considering the number of different type variables occurring in θ. On each substitution, this number is decreased by one (due to the Circularity Clause). Since it cannot decrease below 0, the Algorithm will, after some time, no longer call for substitutions to be performed. The forms under consideration will then remain fixed, and completing their traversal is further clearly a finite process.

3.3. Using the Unification Algorithm to unify polytypes

The Unification Algorithm can be used to determine $\pi' \cap \pi''$ for two arbitrary given polytypes π' and π''. The Algorithm will not immediately do, because type variables occurring in both of the forms to be unified are simultaneously subject to the substitutions performed. This means in particular—using as example the partial type requirements for the PRODUCE command treated earlier—that it unifies

$$(<\text{NUMBER}>, \alpha_1, \alpha_2) \text{ and } (\alpha_1, \alpha_2, <\text{NUMBER}>)$$

to

$$(<\text{NUMBER}>, <\text{NUMBER}>, <\text{NUMBER}>),$$

instead of to

$$(<\text{NUMBER}>, \alpha_1, <\text{NUMBER}>).$$

Further unification with $(\alpha_1, <\text{TABLE}, \alpha_2, \alpha_3>, \alpha_2)$ is patently

impossible. If π' and π'' have no type variables in common, then, as we shall see, the unification produced (if any) is $\pi' \cap \pi''$. This is helpful, since the variables in a polytype are (implicitly) bound and have no inherent meaning. Without change of meaning to Types(π''), we may take any type variable occurring in π'' and replace it systematically by a new variable not occurring in π'. In this way, the sets of variables in π' and π'' can be made disjoint.

So assume that π' and π'' have no common variables. First, we wish to establish that the condition, for some monotype ϕ,

$$\exists \Theta', \Theta'' : \phi = \pi'\Theta' = \pi''\Theta''$$

is then equivalent to

$$\exists \Theta''' : \phi = \pi'\Theta''' = \pi''\Theta'''.$$

In one direction this is trivial: a unification of π' and π'' is also a common refinement of these two. For the other direction, we have to show that a common refinement ϕ of π' and π'' is also a unification. If we restrict Θ' to substitute only for variables occurring in π'—which has clearly no effect on the outcome of $\pi'\Theta'$—, and do similarly for Θ'', then Θ' and Θ'' substitute for disjoint sets of variables, and Θ' does not substitute for variables in π'', nor Θ'' for variables in π'. The substitution Θ''' consisting of the *simultaneous* application of Θ' and Θ'' is then such that $\pi'\Theta''' = \pi'\Theta'$ ($= \phi$) and $\pi''\Theta''' = \pi''\Theta''$ ($= \phi$). So we have obtained ϕ as unification of π' and π''. (Note that Θ''', $\Theta'\Theta''$ and $\Theta''\Theta'$ may all three be different, since the variables occurring on the right-hand side of, e.g., Θ', may still occur on the left-hand side of Θ''.)

Using this equivalence, the condition $\pi = \pi' \cap \pi''$ for π' and π'' with disjoint variables reduces to: a monotype ϕ is a refinement of π iff it is a unification of π' and π''. That the most general unification satisfies this condition, follows immediately from the definition.

We know now that the unification produced by the Unification Algorithm, applied to π' and π'' with disjoint variables, *if one is found*, must be $\pi' \cap \pi''$. But will one be found? If they have a common refinement, it is, by the above equivalence, also a unification. We have already seen that the Unification Algorithm will determine then that π' and π'' are unifiable. The answer is of course "No" if π' and π'' have no common refinement. But, with the introduction of <*>, we can consider this a common pseudo-refinement, and we can modify the Unification Algorithm accordingly. This requires three modifications: If one of the two forms to be unified is <*>, the other is superseded by <*>, as well as its occurrences in the forms under consideration. (No error should be reported then: it has already been reported when the first occurrence of <*> originated.) If an atom clash is found, the clashing forms are both likewise replaced by <*>. If a circularity is detected in a substitution $[\![\pi \rightarrow \alpha_i]\!]$, π is replaced by <*>, and instead the substitution $[\![<*> \rightarrow \alpha_i]\!]$ is performed.

So, in summary, in order to determine $\pi' \cap \pi''$, we have to do the following. First eliminate any common variables from π' and π'' by replacing these by fresh variables. Apply next the Unification Algorithm to π' and π'', as modified to cater for <*>. The unification produced is $\pi' \cap \pi''$.

The reader may, perhaps, wonder if a substantially simpler method than the Unification Algorithm would not do, as we happen to know—indeed, take some effort to ensure—that the polytypes to be unified share no variables. This is not the case. To determine, e.g., $(\alpha_1, \alpha_1) \cap ((\alpha_2, \alpha_2), (\alpha_2, \alpha_3))$, the Unification Algorithm goes through intermediate stages in which the forms to be unified do have common variables, and this is essential to its functioning, and for the purpose to which we have put it. In fact, it is easy to see that a simpler algorithm for determining $\pi' \cap \pi''$ can be transformed without effort into a (simpler) unification algorithm.

4. COMPUTING THE TYPE REQUIREMENTS

4.1. Computing local type requirements

An assumption underlying efficient polymorphic type checking is that the programming language is such that for all predefined constructions and operations, the local type requirements can be expressed as a polytype. *B* has been constructed in such a way that this is the case, although it is necessary to twist its type system somewhat to make it fit; see section 6. If a language has not been designed with an eye to the requirements of type polymorphism, it is likely to contain constructions that make polymorphic type checking intractable—if decidable at all—and thereby impractica(b)l(e). An example from an existing programming language: in the ALGOL 68 expression `(s*t)+"a"`, the type requirement on `s` and `t` cannot be brought into the desired format if `*` and `+` are predefined operators from the standard-prelude, for either the type of `s` is some 'PREFSETY1 (row of) character' and that of `t` is some 'PREFSETY2 integral', or vice versa.

The transformation described in section 1.2 for function definitions into command definitions can also be used to cater for the predefined functions in *B*. For example, the command

```
WRITE a^^(b+c)
```

writes the `(b+c)`-fold repetition of the text `a` to the screen. It can be transformed into

```
ADD b AND c GIVING tmp1
REPEAT a TIMES tmp1 GIVING tmp2
WRITE tmp2.
```

By this transformation, `tmp1` and `tmp2` have become local variables. The external type requirements for the pseudo-commands ADD and REPEAT are known from the language definition. Computing local type requirements as described in section 2.2 would give long tuples, as a position would have to be assigned for each temporary introduced by the transformation. But as we know that these have only a purely local significance, we can locally combine the type requirements of the commands resulting from the transformation (three in the above example) and next take out the positions corresponding to temporaries (similar to the way in which external type requirements are obtained from internal ones).

In a block-structured language—which *B* is not—blocks can be handled in the same way.

4.2. Computing the type requirements for a single unit

Using the machinery developed, the polymorphic type checking, applied to a single unit, proceeds as follows.

- Collect from all constructions in the unit the local type requirements on the type ϕ of the formal parameters and variables occurring in the unit.
- Express each of these as a polytype π_i (signifying the requirement $\phi \in \text{Types}(\pi_i)$).
- Compute the internal type requirement $\pi = \bigcap_i \pi_i$.
- Obtain the external type requirement from π by "projecting" π on the formal parameters, i.e., by removing from (the tuple that is) π the components corresponding to local variables.

To apply this to a unit U, we need to know, beforehand, the type requirements for all units invoked in U. This is a problem, especially if one or more of the units are recursive. Apparently, it is necessary to compute the type requirements for a set of units simultaneously.

4.3. Computing the type requirements for a set of units

We introduce the notion of units being "up for investigation". The following algorithmic description is then almost an algorithm for computing, by successive approximations, the type requirements for a set of units. Almost, for the process is not guaranteed to terminate.

"Almost Algorithm" (*AA*):

- Put all units up for investigation, and set, initially, for each unit U, its internal type requirement to $(\alpha_1, \ldots, \alpha_n)$, creating a type variable for each formal parameter and local variable occurring in U. (This polytype imposes no undue requirement, since $\text{Types}(\alpha_1, \ldots, \alpha_n)$ contains all monotypes that are tuples of the right length.)
- While there is some unit U up for investigation:
  - Compute (a new approximation of) its type requirement, using the current (most recently computed) external type requirements for units invoked in U.
  - U is no longer up for investigation.
  - If the newly computed external type requirement of U differs from the type requirement it supersedes (not counting trivial changes by the renaming of variables), put all units that invoke U up for investigation.

If none of the units considered is recursive, this process will terminate. This can be seen by induction on n, the number of units. If $n > 0$, there must be some unit at the bottom of the calling hierarchy. Sooner or later it will be investigated, for it is up for investigation, and we know by the inductive hypothesis that the process terminates when confined to the other $n-1$ units. Once that unit has been investigated, it can not reappear for investigation, as it contains no invocations of other units. The process then stays confined to the other $n-1$ units and terminates. This termination argument suggests that it is possible to (mis)use the freedom in choosing the next unit to be investigated in such a way that the number of turns can be up to 2^n-1. It would be sensible to perform first a "topological sort" on the units, so that definitive type requirements are computed in one sweep over the units. In the general case, with recursion, this is impossible. A simple improvement is never to re-investigate a unit U while there are still other units up for investigation that were already so at the last investigation of U, and have not been investigated since.

Much work may be duplicated if a unit is re-investigated. This issue would be easier to treat after having laid the groundwork for incremental checking, but in view of the development ahead we need an amendment to *AA* now. Since the internal type requirement of a unit is computed as $\pi = \bigcap_i \pi_i$, and a change in a contributing π_k to π_k' results from a refinement to π_k—so $\pi_k' = \pi_k \cap \pi_k'$—, the new internal type requirement π' resulting from that change equals $(\bigcap_i \pi_i) \cap \pi_k' = \pi \cap \pi_k'$. The amendment is now that recomputations in re-investigations in *AA* indeed take this form: the internal type requirement of a unit U is updated by unifying it with the refined local type requirements stemming from invoked units that caused U to be up for investigation.

This amendment—apart from the tremendous improvement in efficiency—makes it possible to keep track of the type variables occurring in the process, and this gives a hold for tackling recursion, which is allowed and may thwart the termination of the process.

A simple example of a *B* unit for which the process, as described by *AA*, does not terminate, is given by

```
HOW'TO R x:
    PUT x IN y, y
    R y.
```

Let us go through some steps. Let ξ and η correspond to `x` and `y`, respectively. Initially, the internal type requirement, with $\phi = (\xi, \eta)$, is (α_1, α_2), which, projected on ξ, gives α_1 for the external type requirement of the `R` command. The local type requirement derived from the `PUT` command, is $((\alpha_1, \alpha_1), \alpha_1)$, and that of the invocation of `R` is (α_1, α_2). Combining these gives the new internal type requirement $((\alpha_1, \alpha_1), \alpha_1)$. The external type requirement becomes now (α_1, α_1). This is different from the previous requirement, so another computation step is taken. Because of the amendment, only the local type requirement for `R` has to be considered. It is now $(\alpha_1, (\alpha_2, \alpha_2))$. Combining this with the previous internal type requirement, followed by projection, gives the new external requirement $((\alpha_1, \alpha_1), (\alpha_1, \alpha_1))$. The next turn will give us $(((\alpha_1, \alpha_1), (\alpha_1, \alpha_1)), ((\alpha_1, \alpha_1), (\alpha_1, \alpha_1)))$, and so on, in ever increasing complexity. The process tries, as it were, to compute a solution of an equation $\pi = (\pi, \pi)$, but as this solution cannot be expressed by a finite polytype, it can do no better than come up with more and more refined approximations.

The first thing to do now, is to show that if there *is* a solution, the process will find it. This can be proved completely formally, but hopefully a sketch of the crucial idea will suffice. Since the process never draws a conclusion that it is not forced to, the approximation of the type requirements is at all stages such that it can still be refined to the solution, if any. This puts an upper bound on the complexity of the computed approximations: if π can be refined to π', the length of the expression π is at most that of π'. (This is easily proved by induction on the structure of polytypes.) However, each time the process finds a different (approximation of the) type requirement for a unit, the new type requirement is a proper refinement of the old one, and so is longer. So, if the process does not terminate, it will after some time reach an approximation that is too complex to be refinable to the solution, which is absurd if a solution exists.

Next, we turn to the question of whether the absence of a solution really means that there is something wrong with the units (not with the process). For maybe the expressive weakness of polytypes is to blame: maybe there are sets of monotypes, one for each unit, such that a (symbolic) computation, started on any unit with any monotype chosen from its set as the type of its actual parameters, invokes other units, again only with actual parameters whose monotypes belong to their sets. These sets would then form a solution, but maybe there are no polytypes expressing these sets. However, the above argument using the complexity of approximations still goes

through if we concentrate on one unit whose type requirement grows unboundedly—and if the total grows beyond all limits, there must be at least one unit for which this is the case—and then for "solution" above, read: some monotype acceptable as actual-parameters type for that unit.

The type violation in the above unit R is not very different from that in

```
HOW'TO S x:
    PUT x IN x, x.
```

In this case, the error will be found by the Circularity Clause of the Unification Algorithm. In the case of R, it "slips through", because the problem lies in the future use, and type requirements are not propagated forwards, but only backwards (i.e., from invocation to invoker). It would be caught if, before applying the Unification Algorithm to determine $\pi' \cap \pi''$, the variables in both polytypes did not have to be made disjoint.

But note that the following is entirely acceptable (as far as polymorphic type checking is concerned):

```
HOW'TO T x:
    PUT x, x IN y
    T y.
```

The process will terminate here (with success) after the first turn. This is a difference between the polymorphic type checking presented here, and the theory given by MILNER[10]. There, the actual parameters in a recursive invocation are required to have the same polymorphic type as the formal parameters, and a proper refinement is not acceptable. A similar restriction for *B*, although probably acceptable from a pragmatic point of view, is not simple to phrase without going deeper into the mysteries of type polymorphism than is acceptable for the user community for which *B* is intended. Also, it would not make the type checking method described here much simpler: some parts would become much more complicated, possibly more so than the other parts are simplified.

4.4. Detecting non-termination

If we now had some general method for determining non-termination, we could turn *AA* into an algorithm, for if it is found not to terminate, an error may be reported, and the process halted. Such a method exists, and will now be presented.

Consider the type requirements after each unit has been investigated at least once. Call the type variables occurring in these requirements "ancestor variables". Each type variable occurring in the process thereafter is either one of the ancestor variables, or is obtained by "refreshing" an already existing type variable in order to make the Unification Algorithm compute some $\pi' \cap \pi''$ in the recomputations. Thus, each type variable occurring in the process may be thought of as being a "variant" of one of the ancestor variables. It is possible to keep track of the ancestorship of these variants by initially tagging each ancestor variable with a unique tag, and subsequently keeping the ancestor tag when refreshing a variable. Since there is a one-to-one correspondence between ancestor variables and tags, the distinction between these two is dropped, and we will simply speak about "ancestors".

Also, each refinement during the process consists of a substitution in one or more polytypes "under consideration" in the Unification Algorithm. In each such substitution step, at least one of these polytypes has resulted from a (possibly empty) succession of refinements, applied originally to some ancestor, and then to variables contained in the result. These polytypes are said to be "descendants" of the original ancestors from which they resulted. (It is easily seen that different instances of a given ancestor have identical descendants at each stage between recomputations. Although no essential use is made here of that fact, it allows us to speak about "the" descendant of an ancestor. In a practical implementation, this fact is of course exploited, using the ideas of BOYER & MOORE[2].)

Let us apply this to the example of the unit R in the previous section. After the first investigation, the internal type requirement of R is $((\alpha_1, \alpha_1), \alpha_1)$. So there is one ancestor, α_1, and by tagging it with a prime, we obtain $((\alpha_1', \alpha_1'), \alpha_1')$. The external type requirement for R commands, obtained by projection on the ξ coordinate, gives (α_1', α_1'). If we now continue the execution of *AA*, we only have to consider the invocation of R. The local type requirement for the invoked R becomes $(\alpha_1, (\alpha_2', \alpha_2'))$. Here, α_2' is a variant of the original α_1'. Unification now causes the substitution $[\![(\alpha_2', \alpha_2') \rightarrow \alpha_1']\!]$ to be applied to the polytype α_1', which is the ancestor α_1', and so is a descendant of it. So we can see that we are applying a substitution that introduces a variant of α_1' within a refinement of α_1'. This means that the final solution π to which α_1' would be refined, if such a solution exists, has to satisfy some equation of the form $\pi = <..., \pi\Theta, ...>\Theta'$. It is clear that this equation has no solution in finite polytypes.

This observation can be turned into a criterion for deciding that *AA* is going into an infinite loop. Let *CV* ("contains variant of") be a set of pairs of ancestors, interpreted in the usual way as a binary relation. *CV* may be visualized as the set of edges of a directed graph whose nodes are the ancestors. Instead of $(a, a') \in CV$, we write $a \rightarrow_{CV} a'$. Initially, *CV* is empty. If a substitution $[\![\pi \rightarrow \alpha_i]\!]$ is applied to a descendant of an ancestor a, then for all ancestors a' of variants properly contained in π, we add $a \rightarrow_{CV} a'$ to *CV*. We can formulate now a second circularity clause:

> If, at some time, $a \rightarrow_{CV} a'$ is added, but this addition causes $a \rightarrow^+_{CV} a$ to hold, where $\rightarrow^+_{CV}$ stands for the transitive closure of $\rightarrow_{CV}$:
>
> It is determined that *AA* will not terminate, a type violation is signalled, and the execution of *AA* is halted.

Regarding the correctness of this criterion, two things must be shown. The first is that indeed, if a cycle is found in *CV*, there is a loop in *AA*. Let $\pi \twoheadrightarrow \pi'$ hold iff π can be expressed in the form $<..., \pi'\Theta, ...>\Theta'$. Clearly, $\twoheadrightarrow$ is transitive. Assume termination of *AA*, and let π and π' be the eventual solutions for ancestors a and a', respectively. If $a \rightarrow_{CV} a'$, then $\pi \twoheadrightarrow \pi'$. This carries over to $\rightarrow^+_{CV}$. If a cycle is found in *CV*, this means that the existence of a solution implies the existence of a polytype π such that $\pi \twoheadrightarrow \pi$. As above in the examination of R, this is impossible for a finite polytype.

Next, if no cycle is found, we want to show the termination of *AA*. Since the number of pairs of ancestors is finite, *CV* must converge to a limit in the process within a finite number of steps. The absence of cycles means that the ancestors may be topologically sorted, where $a \rightarrow_{CV} a'$ implies that a comes before a' in the sorting order. Take the first ancestor a_1 in that order. Since, for no a, $a \rightarrow_{CV} a_1$, either no substitution is ever applied to a_1, or it is refined to a monotype. So after a finite number of steps it has converged to its final limit. For the next ancestor a_2, no refinement can result in a descendant containing variants with other ancestors than a_1, for if a is the ancestor of a variant in a descendant of a_2, we have $a \rightarrow^+_{CV} a_1$, meaning that a comes before a_2. Once a_1 has reached its limit, again either no further substitution takes place in the descendant corresponding to a_2, or the variants of a_1 are replaced by

that limit. So this descendant also converges in a finite number of steps. In general, once the descendants for a_1 to a_{i-1} have reached their limits, that for a_i will also reach a limit. If all descendants have reached their limits, *AA* terminates.

This termination argument establishes also an upper bound on the number of "sweeps" over all units before a loop is detected (if *AA* has not already terminated). For it is easily seen that, if the external type requirements change, each sweep adds an element to *CV*. (And if they do not change, *AA* has terminated.) If the number of elements is equal to that of the ancestors, a cycle must be present in *CV*. The number of ancestors is at most the total n of the lengths of the polytype tuples expressing the type requirements for the units. This upper bound could be used to detect looping of *AA*. However, it is very crude: n may be large; although examples may be constructed in which this upper bound is reached, these are contrived. The detection of cycles in *CV*, e.g., by a depth-first search, is much faster, since cycles resulting from accidental errors typically involve only a few (one or two) ancestors.

5. INCREMENTAL TYPE CHECKING

5.1. Decorating a balanced tree

To report type violations as soon as possible—when they arise during an edit session—the type requirements have to be kept up to date when modifications are made to a unit. The computations as described until now will not do: the time needed, although not prohibitive if a unit is checked as a whole, is too large for an acceptable response time in an editor. So we want to find an organization for the type-checking algorithm that permits the reuse of most of the previously computed information if a small modification is made to a unit. The idea will first be described under the assumption that there is no recursion present. The treatment of recursion will then be resolved as a by-product of the treatment of inter-unit type inconsistencies.

Recall from section 4.2 that the internal type requirement for each unit may be computed as $\pi = \bigcap_i \pi_i$, where the π_i are the local type requirements derived from the constructions contained in the unit. Now the collection of all these local requirements may be organized in the form of a balanced tree. Any of a number of data structures—2-3-trees, AVL-trees, B-trees (KNUTH[7])—will do. The only requirement is that the data structure allows fast algorithms for insertion and deletion. "Fast" means here: time O(log N), where N is the number of items (local requirements) in the collection. The items in the collection may be thought of as labeling the leaf nodes of the tree. (There is such a tree for each unit, so that we have a forest of trees.) Let *PD* ("properly decorated") stand for the assertion: each non-leaf node is labeled with $\bigcap_i \pi_i$, where the π_i, this time, are the items labeling its (immediately descendant) son nodes. Because of the associativity and commutativity of the operation $\cap$, *PD* implies that each node is labeled with the requirement combining all local requirements contained in the sub-tree descending from that node. In particular, the root of the tree is labeled with π, the grand total of all the local requirements.

What we have to do now is to take care that *PD* is at some time properly established, and is further maintained (kept invariant) through all modifications of the tree. Establishing *PD* is easy: at some time we have started with a collection of one item, so that *PD* was met without specific effort. Each local modification can be described as the deletion of the old item, followed by the insertion of the new one. The O(log N) algorithms for insertion and deletion in a balanced tree affect no more than O(log N) nodes. To maintain *PD*, it is sufficient that the type requirement labels are recomputed for the affected nodes only. An immediate further improvement is obtained by observing that the affected nodes typically form a path from the root to the leaf where insertion/deletion takes place. The necessary recomputations may be performed in the order: lowest nodes first, root last. If the modification is locally a refinement of the type requirement—which is so in the important case of an addition to the unit—, it is so throughout the path to the root, so that it is sufficient to unify each label with that of its (just refined) son node. If, at some time, the new label happens to be identical to the old one, the recomputations of labels above may be skipped. Also, the recomputations of a change consisting of a deletion and an insertion may be merged in a natural way. This does not change the worst-case complexity, but greatly reduces the average complexity in practice: in many cases, a modification to a unit will not change its overall type requirement, and the effects of the modification to the local requirements, if any, tend to peter out fast.

In a language such as *B*, a unit already has a tree structure, viz. that implied by the syntax. It is possible to keep this structure if we are willing to give up full balancing of the tree. The balancing would remain for representing collections of nodes that are brothers in the syntax tree. This is acceptable, since the height of a syntax tree tends to be low in *B*; lower, in fact, than O(log N), because of the presence of "refinements". The advantage is that the average complexity is in practice reduced much more if—as is a good idea anyway—descendants in the label of a node stemming from the unification of a descendant in a son node with a completely free polytype α_i are represented in the implementation by a pointer to the son descendant (or by a copy of the pointer representing the son descendant). With some additional effort it is then possible to organize the update process so that changes to the types of *B* variables that are only used locally do not propagate physically to the root—although they do logically through the pointers.

If a type inconsistency is found—a new occurrence of <*> is generated—, it is possible to follow a path from the node where the clash is detected to a leaf node such that that leaf node by itself already clashes with the leaf node just modified. This is helpful in reporting an understandable error message. (This possibility depends on a property that we have not proved, viz., that if $\bigcap_i \pi_i$ contains <*>, then there exists a pair of contributors π_j and π_k such that $\pi_j \cap \pi_k$ already contains <*>.)

In the process of making modifications to a unit, intermediate inconsistent states may be needed to reach a final consistent state. It is undesirable if these are signalled as erroneous. A reasonable approach is to signal inconsistencies (while a unit is still in the process of being edited) only if the part of the unit from its beginning up to the modification is inconsistent. If the modifications are then applied in the textual order and the overall result is consistent, no error will be reported at all. With the balanced-tree representation, this can still be done in time O(log N).

5.2. Inter-unit inconsistencies

The most useful strategy for reporting inconsistencies between different units is not obvious. Potential newly arising inter-unit inconsistencies are of two kinds: the actual parameters of some invocation of the unit being edited no longer fit its modified external type requirement, or the unit contains itself

an invocation with ill-fitting parameters. In either case, a subsequent modification of the other unit may restore consistency.

There are several reasons for not signalling inconsistencies of the first kind immediately when they arise. The first is that the inconsistency is visible only in another unit. Giving a helpful error message is then hard. A second reason falls outside the scope of this paper and is specific to *B*. It is connected to the "static content check" of *B*, which requires that all variables be initialized before use. Here the same two kinds of errors may arise, but it makes little sense to check the first kind with an incomplete unit, so error messages have to be postponed. It would be confusing to the user if the error-reporting strategies were different for these two static checks. Finally, postponing the detection of inconsistencies of the first kind makes it possible to address recursion in a natural way.

The desired postponement can be achieved as follows. At the onset of editing a unit *U*, the nodes corresponding to invocations of *U* are (virtually) deleted from all trees. This may also be interpreted as resetting the external type requirement of *U* to the most general one possible. This cannot cause inconsistencies, of course. It can cause the external type requirements of invoking units to change, which necessitates further recomputations, and so on. Although this may take some time, this is acceptable at the start and end of an edit session. When the editing of *U* is finished, the invocations of *U* are re-inserted at the positions where they were deleted, and another round of recomputation takes place. Inconsistencies then found are reported.

This solves all problems with recursion, since the temporary deletion of *U* invocation nodes effectively cuts off all recursion in which *U* is involved. Note, even, that the virtual deletions have no effect observable by the user, unless the unit where they are applied and *U* are mutually (possibly indirectly) recursive. Thus it would seem possible to skip recomputing type requirements for the other units invoking *U* at the start of the edit session and to recompute only at the end. This is not the case, however, because the modifications to *U* may introduce recursion where it did not exist before.

5.3. Variant strategies

Other strategies can be formulated that are more or less satisfactory, depending on how much effort may be spent on the computations.

An interesting variant is to maintain for the units, in parallel with the type requirements as discussed, "weak" type requirements obtained by disregarding invocations of user-defined units. During the modification of a unit, only the weak requirements are kept up to date. At the end of the edit session, the strong requirements are computed. The advantages are that no special effort is needed at the onset of editing, that the final recomputation may be skipped if the internal weak type requirement of the unit just edited has not changed, and that, failing that, the recomputation rounds on all units are not needed if its recomputed (strong) external type requirement remains the same. A disadvantage is that less type violations are detected on the spot.

Another variant is to maintain, in parallel, type requirements under the assumption that no essential use is made of polymorphism. This means that no variants are made of ancestors in computing local type requirements. These requirements are then, of course, too stringent. A type violation in the normal requirements is reflected in a type violation in the overstringent requirements (but not the other way around). The advantage is that they can be computed without iteration, and that, in practice, the overstringent requirements will often be met. The recomputation rounds for the normal requirements can be postponed until a type violation is found in the overstringent requirements.

5.4. Complexity

A theoretical discussion of the complexity of the algorithm is only possible if the implementation of the Unification Algorithm is taken into account, which will not be done here. Also, the worst-case complexity is not very interesting: it is possible to construct units in which a small modification entails a major upheaval of the type requirements, but this does not correspond to normal programming practice. A practical indication of the complexity is the number of ancestor variables occurring in the decorated tree corresponding to a unit. If that number is A and the size of the unit is N, the time required for a modification will be approximately proportional to $A \log N$ if the change propagates to the root. The values of A were determined for a collection of over one thousand units, collected from user programs, and the least-squares fit for formulae of the form $cN^p (\log N)^q$ was computed. This gave $p = 0.0032$ and $q = 1.63$. (This suggests $p = 0$, but this is theoretically implausible. On *a priori* grounds, one would expect, rather, $cN / \log N$ as estimator for A, but this gave a definitely worse fit.)

6. FITTING *B* TO POLYMORPHISM

6.1. Genericity of texts, lists and tables

Although *B* was designed to allow polymorphic type checking, its type system does not fit the theory as smoothly as suggested until now for the sake of simplicity. The major complication stems from the fact that some functions in *B* can be used generically on texts, lists and tables, and that the notation { } may be used both for an empty list and for an empty table.

In order to fit these types to the structure as required in section 2.1, and allow the necessary latitude to accommodate the additional genericity in terms of type polymorphism, we have to model these in a more complicated way. Instead of <TEXT>, <LIST, α_1> and <TABLE, α_1, α_2>, we can use, respectively,

<TLT, <TX>, <TX>, <NIL>, <TEXT>>,
<TLT, <LT>, <LIST>, <NIL>, α_1> and
<TLT, <LT>, <TABLE>, α_1, α_2>.

This achieves the desired effect. ("TLT" stands, of course, for "text, list or table", "TX" for "text" and "LT" for "list or table".)

A—theoretically equivalent—alternative is to leave the types of *B* as given in section 2.1, but to add two half-baked types

<TLT, α_1> and
<LT, α_1>,

and to add rules for computing $\cap$ if such types are involved, like <TABLE, π_1, π_2> $\cap$ <TLT, π_3> = <TABLE, $\pi_1, \pi_2 \cap \pi_3$>, and <TEXT> $\cap$ <TLT, π_1> = <TEXT> $\cap$ π_1 (but <TEXT> $\cap$ <LT, π_1> = <*>).

6.2. Circularity of <TEXT>

B has one circular type, viz. that of texts. Selection on a text does give a character, but the type of that value is the same as that of other texts. This means that the two circularity clauses have to be amended, to check first if the circularity detected fits the circular pattern of the type <TEXT> $\subset$

<TLT, <TEXT>>. A simple way to do this is, on a substitution $[\![\pi \rightarrow \alpha_i]\!]$ that would cause a circularity error, to unify π and <TEXT> instead, and next apply $[\![<\text{TEXT}> \rightarrow \alpha_i]\!]$. Care has to be taken to issue an appropriate error message if a type incompatibility occurs in the unification, since this means in reality that the circularity was unlawful.

7. CONCLUSION

7.1. Relation to other work

The work reported here differs from other work on inferring types, in that an organization is described that allows the incremental computation of type requirements.

Also, no other authors allow recursion to be combined with true type polymorphism. Although this is not of practical importance as far as the usability of *B* is concerned—only with great effort is it possible to define useful units using this possibility —it helps to keep the abstract semantics of the type requirements simple. These can be formulated as: arbitrarily deep "macro expansion" does not exhibit (monotypical) type violations. This is also the meaning of the notion of "formal correctness with respect to parameter transmission" used by LANGMAACK[8]. In this sense, the method presented is complete, meaning that a formally correct set of units passes the type checks. The notion of "completeness" used by MILNER[10] and DAMAS & MILNER[3] is weaker: it is defined with respect to the existence of a "well-typing" or "principal type-scheme". It is easy to give examples that cannot be well-typed in the sense of MILNER, but that are formally correct in the sense of LANGMAACK.

Note, however, that our stronger completeness is possible only by virtue of "weaknesses" of *B*: no infinite types and no procedures as parameters. This explains why the argument of LANGMAACK exhibiting the undecidability of formal correctness if parameter specifications are not complete does not apply to *B*. (The undecidability result of GEHANI[5] does not refer to type inference at all. Moreover, it uses a weaker requirement than formal correctness. If formal correctness were required, a similar method for detecting non-termination as described in section 4.4 could be used to obtain a decision procedure.)

The work of SUZUKI[12] is an extension of the methods of MILNER[10] for type inference in Smalltalk. Next to some innovations that are not needed for *B*, in particular for treating union types, and the treatment of recursion, a difference is that the inferred types are not used for strong type checking, since Smalltalk is not strongly typed. In the system proposed for Smalltalk by BORNING & INGALLS[1], parameters are specified, which makes the problem to be solved incomparable to that addressed by the present work. The same is true for the type checking method of DEMERS & DONAHUE[4] for Russell. Moreover, their notion of type consistency differs from the one used here: the requirements are propagated forwards, rather than backwards.

7.2. Further research

The system given here is complete, and this was possible because of the simplicity of *B*. That this simplicity plays a role follows from the undecidability result of LANGMAACK[8]. It would be of interest for the design of future languages using type polymorphism, to have a clearer delineation of the conditions under which completeness can still be obtained. For example, the question is open whether arbitrary recursive types can be allowed if procedure parameters are still excluded (or the other way around).

Another interesting issue is to see if, and how, user-defined data types can be combined with type polymorphism, retaining both static type checking and freedom from the duties of declaration and specification.

Acknowledgements

I am grateful to Leo Geurts, Dick Grune, Jan Heering, Paul Klint and Timo Krijnen for their unabating willingness to listen to my rambling, ranting and raving during the various stages of this research, and especially for their insistence that there had to be a simpler way of detecting recursive circularities than that devised until then; and to Steven Pemberton for his close reading of an earlier version.

REFERENCES

[1] BORNING, A.H. & D.H.H. INGALLS, A type declaration and inference system for Smalltalk, Conf. Record 9th ACM Symp. Principles of Programming Languages, 133-141, 1982.

[2] BOYER, R.S. & J.S MOORE, The sharing of structure in theorem proving programs, Machine Intelligence **7**, 101-116, (B. Metzer & D. Michie, eds.), Edinburgh University Press, 1972.

[3] DAMAS, L. & R. MILNER, Principal type-schemes for functional programs, Conf. Record 9th ACM Symp. Principles of Programming Languages, 207-212, 1982.

[4] DEMERS, A.J. & J.E. DONAHUE, Data types, parameters and type checking, Conf. Record 7th ACM Symp. Principles of Programming Languages, 12-23, ACM, 1980.

[5] GEHANI, N., Generic procedures: an implementation and an undecidability result, Comp. Languages 5 (1980) 155-161.

[6] GEURTS, L.J.M. & L.G.L.T. MEERTENS, Designing a beginners' programming language, New Directions in Programming Languages 1975, 1-18, (S.A. Schuman, ed.), IRIA, Rocquencourt, 1976.

[7] KNUTH, D.E., The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, 1973.

[8] LANGMAACK, H., On correct procedure parameter transmission in higher programming languages, Acta Informatica **2** (1973) 110-142.

[9] MEERTENS, L.G.L.T., Draft Proposal for the *B* Programming Language—Semi-Formal Definition, Mathematical Centre, Amsterdam, 1981.

[10] MILNER, R., A theory of type polymorphism in programming, J. Computer and System Sciences **17** (1978) 348-375.

[11] ROBINSON, J.A., A machine-oriented logic based on the resolution principle, J. ACM **12** (1965) 23-41.

[12] SUZUKI, N., Inferring types in Smalltalk, Conf. Record 8th ACM Symp. Principles of Programming Languages, 187-199, 1981.

[13] TENENBAUM, A.M., Type determination for very high-level languages, Comp. Sci. Rep. NSO-3, Courant Institute of Mathematical Sciences, NYU, New York, 1974.