Formal Aspects of Computing (1996) 8: 565–584

Formal Aspects of Computing

Richer Types for Z

Michael Spivey
Oxford University Computing Laboratory, Oxford, UK

Keywords: Type system; Specification language Z

Abstract. The formal specification language Z is strongly typed, but has a rather inexpressive type system in which essentially the only type constructors are power set and Cartesian product. This paper explores the possibility of using a richer type system for extracting information about a Z specification, so that properties like being a function or a sequence become part of the type of an expression. This richer type system adds further type constructors to the sparse set contained in the language definition, and uses properties of Z's library functions to infer information about complex expressions from information about their simpler parts.

1. Introduction

The Z language [Spi88, Spi92a, Nic95] is strongly typed, in the sense that the rules of the language assign a type to each expression in a Z specification, and specifications are not considered meaningful unless a consistent assignment of types to expressions is possible. The type system of Z is rather weak, however, and properties like being a mathematical function or being a sequence are not reflected in the type of a binary relation.

In a well-formed Z specification, each expression has a type that is formed from the basic types of the specification by applying three type constructions: the power set construction ($\mathbb{P}$), n-fold Cartesian products ($\cdots \times \cdots \times \cdots$), and a 'schema product' construction that builds record types associated with certain operations on schemas. This small collection of type constructors is sufficient to assign a type to every object that occurs in the traditional development of discrete mathematics based on set theory, and the coarse nature of the type system allows

Correspondence and offprint requests to: Dr J. M. Spivey, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK.

familiar mathematical concepts to be defined in the traditional way. However, it means that the type of an expression as computed by a type checking algorithm contains less information than the 'type' that is ascribed to the expression in informal discourse about a specification.

For example, in Z the set of all (partial) functions from X to Y, written $X \pfun Y$, is actually equal to the set of relations between X and Y that map each element of the domain to exactly one element of the range:

$$X \pfun Y = \{f : X \rel Y \mid (\forall x : X; y_1, y_2 : Y \bullet (x, y_1) \in f \land (x, y_2) \in f \Rightarrow y_1 = y_2)\}$$

The type of each side of this equation is $\mathbb{P}(\mathbb{P}(X \times Y))$, that is, the type of a set of relations between X and Y. An individual function or relation would have type $\mathbb{P}(X \times Y)$: in Z, as in conventional mathematics, a function is simply a set of ordered pairs. As another example, a sequence over a set X is modelled in Z as a function from some interval $1 \mathinner{..} n$ to X, and the set $\mathsf{seq}\, X$ of all such sequences is defined as follows:

$$\mathsf{seq}\, X = \{f : \mathbb{N} \pfun X \mid (\exists n : \mathbb{N} \bullet \mathsf{dom} f = 1 \mathinner{..} n)\}$$

Each sequence has type $\mathbb{P}(\mathbb{Z} \times X)$, and the set of all sequences over X therefore has type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times X))$.

Whilst such examples demonstrate that a type system based on $\mathbb{P}$ and $\times$ is sufficient to allow the construction of the standard models of mathematical concepts, they also illustrate the conceptual gap between the types assigned to expressions by the Z type system and the informal way human readers and writers think of the same expressions. Fluent readers of Z specifications know that sequences are a special kind of function, and that they can be treated as functions where necessary: for example, the second element of a sequence s can be extracted by writing $s(2)$, applying the sequence as a function to the argument 2. But they do not think exclusively in these terms: for example, it is much easier to identify the concatenation operator on sequences (written $\frown$ in Z) as belonging to the set

$$\mathsf{seq}\, X \times \mathsf{seq}\, X \fun \mathsf{seq}\, X$$

than to recognize that its type is

$$\mathbb{P}((\mathbb{P}(\mathbb{Z} \times X) \times \mathbb{P}(\mathbb{Z} \times X)) \times \mathbb{P}(\mathbb{Z} \times X))$$

Software tools that process specifications and report the types of expressions (as type checkers do) are much easier to use if they report the 'type' of concatenation in something close to the first of these forms rather than in the second form, even if the second form is (strictly speaking) the correct type. The subject of this paper is a method for systematically deriving 'enriched types' of this kind, with a certain guarantee of accuracy. This method is implemented in the author's *fuzz* type checker, allowing it to report the types of expressions in a form that is easier for its users to understand. The method actually constitutes an alternative type system for Z specifications, richer than the 'official' type system based on $\mathbb{P}$ and $\times$.

A type checking program must report as errors only those situations where the actual types of expressions fail to match, even if the enriched types are different. It is *not* an error to treat a sequence as if it were a function, because sequences actually are functions in Z, and part of the power of the notation comes from the facility to use functional operations on sequences. Thus a type checking program

that uses the enriched type system for calculating the types of expressions must have a way of expanding type abbreviations, in order to derive the official type of each expression. It is when these official types fail to match that an error should be reported, even if the error message explains the error in terms of enriched types.

2. Typing Rules

The rules for assigning types to expressions in Z can be expressed as inference rules for deriving typing assertions of the form $E :: t$, where E is a Z expression, and t is a type. As examples, we give here the rules for ordered pairs $(E_1, \ldots, E_n)$, for Cartesian products $E_1 \times \cdots \times E_n$, for set displays $\{E_1, \ldots, E_n\}$, and for function application $E_1(E_2)$. The typing rules depend on an *environment* that gives the types of the variables that may appear in the expression, but we omit the details here, since they are not relevant to the subject of this paper.

An n-tuple $(E_1, \ldots, E_n)$ may have components $E_1, \ldots, E_n$ of any types, and the tuple itself has a Cartesian product type constructed from the types of the components. This is expressed in the following typing rule:

$$\frac{E_1 :: t_1 \quad \ldots \quad E_n :: t_n}{(E_1, \ldots, E_n) :: t_1 \times \cdots \times t_n}$$

Like an inference rule in logic, this rule allows a conclusion to be derived from certain premisses. Here both the conclusion and each premiss assert that a certain expression has a certain type. Rules like this allow the type of a complex expression (in this case, an ordered n-tuple) to be deduced from the types of simpler sub-expressions.

The arguments $E_1, \ldots, E_n$ of a Cartesian product $E_1 \times \cdots \times E_n$ are required to have power set types $\mathbb{P}\, t_1, \ldots, \mathbb{P}\, t_n$; if the arguments have such types, then the whole expression has a type constructed from the types $t_1, \ldots, t_n$:

$$\frac{E_1 :: \mathbb{P}\, t_1 \quad \ldots \quad E_n :: \mathbb{P}\, t_n}{E_1 \times \cdots \times E_n :: \mathbb{P}(t_1 \times \cdots \times t_n)}$$

The value of a Cartesian product is a set of tuples: for example, the value of *even* × *even* is the set of all pairs of numbers that are both even; its type, $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$ is the type that contains all sets of pairs of numbers.

In a set display $\{E_1, \ldots, E_n\}$, all the expressions E_i must have the *same* type; the type of the whole expression is that of a set of objects of this type:

$$\frac{E_1 :: t \quad \ldots \quad E_n :: t}{\{E_1, \ldots, E_n\} :: \mathbb{P}\, t}$$

As a final example, function application $E_1(E_2)$ has a typing rule that requires E_1 to have a type $\mathbb{P}(t_1 \times t_2)$, since this is the type of a function from t_1 to t_2. The type of the expression E_2 is required to match the type t_1 expected by the function E_1, and the type of the whole expression is the result type t_2:

$$\frac{E_1 :: \mathbb{P}(t_1 \times t_2) \qquad E_2 :: t_1}{E_1(E_2) :: t_2}$$

Typing rules such as those given here can be implemented using an algorithm that makes a single bottom-up pass over an expression tree[1]. Given a complex expression, the algorithm first computes the types of its sub-expressions; then it applies the unique typing rule associated with the form of the expression, checking that the types computed for sub-expressions have the appropriate form, and generating the type of the whole expression by filling in the pattern given in the rule. If, at any point, the types calculated for sub-expressions do not have the form required for application of the relevant rule, then the expression is not properly formed according to the typing rules. This happens, for example, if one of the arguments of a Cartesian product $E_1 \times \cdots \times E_n$ does not have a power-set type, or if the type of the argument E_2 in an application $E_1(E_2)$ does not match the type expected by the function E_1. Each construct that forms an expression in Z is associated with exactly one typing rule, so the type derived by this algorithm is the only type ascribed to an expression by the rules.

3. Generic Constants

Z has a standard library of mathematical concepts that are used in most specifications, and a strength of the language is that these concepts are built up by making definitions in the language, not by including the concepts as built-in language features. This means that the mathematical library is open-ended, and can be extended with general concepts that are of special use in some application area. Many of the definitions in the standard library are generic, in the sense that they can be used to build sets, relations, sequences, and so on, with elements drawn from any set. The Z language provides notational facilities for making generic definitions, and includes notation for taking a name that is defined generically and applying an instance of it in a particular context. The generic parameters may be provided either explicitly by expressions or implicitly from the context.

For example, here is a definition of the function *first* that extracts the first element of an ordered pair:

$$[X, Y]$$
$$\mathit{first} : X \times Y \longrightarrow X$$
$$(\forall x : X; y : Y \bullet \mathit{first}(x, y) = x)$$

The formal parameters $[X, Y]$ that appear at the top of the box indicate that a function $\mathit{first}[X, Y]$ is being defined here for *each* pair of sets X and Y. The axiom given below the horizontal line states that, whatever sets X and Y are involved, *first* is the function that maps each pair (x, y) to its first element x.

After a generic constant such as *first* has been defined, it can be used in expressions such as $\mathit{first}[\mathbb{Z}, \mathbb{Z}](3, -4)$, where the actual parameters corresponding to X and Y are given explicitly. We can extend our system of type rules to cover this case by adding the following rule, which ascribes a type to expressions of the form $x[E_1, \ldots, E_n]$, where x is a generic constant and $E_1, \ldots, E_n$ are explicit

[1] There are a few expressions, such as the empty set display $\{\}$, that cannot be handled by a simple bottom-up algorithm, but they can be handled by the techniques discussed later for inferring implicit generic parameters.

generic parameters:

$$\frac{E_1 :: \mathbb{P}\, u_1 \quad \ldots \quad E_n :: \mathbb{P}\, t_n}{x\,[E_1, \ldots, E_n] :: t\{u_1, \ldots, u_n / z_1, \ldots, z_n\}} \quad [\, \rho(x) = [z_1, \ldots, z_n] \bullet t \,]$$

In this rule, we assume that the *generic type* $[z_1, \ldots, z_n] \bullet t$ is assigned to x by the environment ρ; this type records the generic parameters $z_1, \ldots, z_n$ of the definition of x and a type t that depends on those parameters. The actual parameters must be sets, and thus have power-set types $\mathbb{P}\, u_i$. The type of the whole expression, $t\{u_1, \ldots, u_n / z_1, \ldots, z_n\}$, is obtained by substituting the types u_i for the formal parameters z_i in the type t.

We can apply this rule to the example *first* $[\mathbb{N}, \mathbb{Z}](3, -4)$. The generic definition shown above results in *first* being added to the environment with type

$$[X, Y] \bullet \mathbb{P}((X \times Y) \times X)$$

This is the type of a generic function from $X \times Y$ to X. The actual parameters that are supplied are $\mathbb{N}$ and $\mathbb{Z}$, and both of these expressions have the type $\mathbb{P}\, \mathbb{Z}$; thus the requirement that the actual parameters have power-set types is satisfied, and the types we should substitute for X and Y are $\mathbb{Z}$ and $\mathbb{Z}$. This gives $\mathbb{P}((\mathbb{Z} \times \mathbb{Z}) \times \mathbb{Z})$ as the type of the applied occurrence *first* $[\mathbb{N}, \mathbb{Z}]$. This function is applied to the argument $(3, -4)$, which has type $\mathbb{Z} \times \mathbb{Z}$. (In Z, functions of several arguments are identified with functions whose single argument is a tuple.) According to the typing rule for function application, this application is well-typed, and gives a result of type $\mathbb{Z}$.

In addition to explicit instances of generic constants, Z also allows the actual parameters to be implicit: thus the expression *first* $[\mathbb{Z}, \mathbb{Z}](3, -4)$ could be abbreviated as *first* $(3, -4)$, and the actual parameters are then types that are determined by the context: in the example, we can determine that these types are $\mathbb{Z}$ and $\mathbb{Z}$, otherwise the type of *first* would not match the type of its argument $(3, -4)$. Note that implicit parameters are always chosen to be types, rather than arbitrary set-valued expressions; this means that they are usually determined uniquely by the context in which a generic constant is used. Implicit parameters are indispensable for readable specifications: without them, even so simple a formula as $\mathsf{dom}\, password \subseteq users$ would have to be written as something like

$$\mathsf{dom}[NAME, PASSWORD]\, password \subseteq_{[NAME]} users$$

and for more complex formulas the weight of formal clutter would quickly become intolerable.

The rule for ascribing types to generic constants with implicit parameters is as follows:

$$\frac{}{x :: t\{u_1, \ldots, u_n / z_1, \ldots, z_n\}} \quad [\, \rho(x) = [z_1, \ldots, z_n] \bullet t \,]$$

This rule states that, when the generic parameters are left implicit, the type of a generic constant x may be obtained by substituting *any* types $u_1, \ldots, u_n$ for the formal generic parameters of x. This rule ascribes many different types to an applied occurrence of a generic constant, so it spoils the property of the typing system that every expression has a unique type. For example, the reference *first* has *every* type of the form $\mathbb{P}((u_1 \times u_2) \times u_1)$. Nevertheless, in the expression *first* $(3, -4)$, there is only one choice of u_1 and u_2 that allows a type to be assigned to the whole expression, and that choice is $u_1 = u_2 = \mathbb{Z}$.

The fact that generic constants can be ascribed many different types means that it is no longer sufficient to compute types in an expression in one bottom-up pass that ascribes a single type to each sub-expression. Instead, we can borrow the techniques that are used in compilers for polymorphic functional languages like ML [Mil78]. So-called type variables are used as markers for types that are as yet unknown, and wherever types must match, unification is used to solve constraints on the type variables and substitute known types for the unknowns. The chief difference (as far as type inference is concerned) between Z specifications and ML programs is that the generic constants of Z are families of monomorphic values, rather than single polymorphic values. This means that, to avoid semantic ambiguity, each type variable must be assigned a single known type by the context. It is easy to test this in a type checker based on unification, simply by verifying that no unassigned type variables remain after the checking of a formula is finished.

Although type variables are a vital ingredient in effective type checkers for Z, they play no part in the language itself, and a specification in Z is well-typed exactly if there is a unique type t for each expression E contained in the specification, and a unique derivation of the typing assertion $E :: t$, so that any implicit parameters are uniquely determined. The correctness of a typing algorithm based on type variables with respect to this definition of the type system is proved in [SpS90]. In this paper, we shall continue to discuss type systems for Z in terms of typing rules that do not involve type variables, pausing from time to time to discuss the implementation of these rules by algorithms that use type variables.

Generic definitions are used in the Z library both to introduce operations like *first*, dom, and $\frown$, and to introduce sets like $X \pfun Y$ and $\mathsf{seq}\, X$. In fact, the partial function symbol $\pfun$ is introduced by a definition of the form

$$X \pfun Y == \{f : X \rel Y \mid \ldots\}$$

This can be regarded as an abbreviation for the definition

$$\begin{array}{|l} [X, Y] \\ \hline \_ \pfun \_ : \mathbb{P}(X \rel Y) \\ \hline (\_ \pfun \_) = \{f : X \rel Y \mid \ldots\} \end{array}$$

The symbol $\pfun$ is an *infix generic symbol* in Z, so the expression $X \pfun Y$ is short for $(\_ \pfun \_)[X, Y]$, that is, the instance of the generic constant $(\_ \pfun \_)$ where the actual generic parameters are X and Y. Similarly, seq is a generic prefix symbol, and the expression $\mathsf{seq}\, X$ stands for the instance $(\mathsf{seq}\, \_)[X]$ where the actual parameter is X. The enriched type system proposed in this paper elevates generic constants like $\pfun$ and seq to the status of type constructors, and allows extra information about generic functions like *first*, dom and $\frown$ to be recorded and used in deducing the types of expressions.

4. Abbreviated Types

The discussion so far has concentrated on the 'official' type system of Z, as described in the language reference manual and adopted in the new draft standard. We now turn to the richer type system that is the true subject of this paper. In this alternative type system, certain generic set-valued constants from a Z specification

are identified as *type abbreviations*. For example, in a specification that includes the standard Z library, the partial function symbol $\pfun$ might be named as a type abbreviation, so that after a declaration

$$f : X \pfun Y$$

the type of f is given as $X \pfun Y$ and not as $\mathbb{P}(X \times Y)$. As a special case, non-generic constant sets like $\mathbb{N}$ can also be named as type abbreviations.

For the enriched type system to work correctly, generic constants that are named as type abbreviations must have the properties of strictness and monotonicity. By *strictness*, we mean the condition that if a is a type abbreviation defined with the type $\rho(a) = [z_1, \ldots, z_n] \bullet \mathbb{P}\, t$, then t should contain at least one occurrence of each parameter z_i. We also disallow a type abbreviation a with $\rho(a) = [X] \bullet \mathbb{P}\, X$. These restrictions are necessary to avoid certain pathological cases: for example, if a type abbreviation *foo* were defined by

$$foo[X, Y] == X$$

then the two types $foo[\mathbb{Z}, \mathbb{Z}]$ and $foo[\mathbb{Z}, \mathbb{P}\,\mathbb{Z}]$ are superficially different but have the same expansion. This complicates both the theory of the type system and the implementation of type checking programs, without adding anything useful to the power of the type system.

By *monotonicity*, we mean that when applied to arguments that are larger as sets, a type abbreviation should deliver a result that is larger as a set. This property holds for the partial function symbol, because if $X \subseteq X'$ and $Y \subseteq Y'$ then $(X \pfun Y) \subseteq (X' \pfun Y')$. It does not hold for the total function symbol $\fun$, because if $X \subseteq X'$ it is not generally true that $(X \fun Y) \subseteq (X' \fun Y)$. Monotonicity of type constructors is necessary for the type system to be sound. For example, suppose we name seq as a type abbreviation, and consider the expression $\mathsf{seq}\, even$. This expression denotes a set of sequences of numbers, so we would like to ascribe to it the type $\mathbb{P}(\mathsf{seq}\,\mathbb{Z})$, in which seq has been used as a type abbreviation; this type abbreviates the official type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}))$. A basic requirement for any sound type system is that the value of an expression (whenever it is defined – but that is a different issue) should be a member of its type: in this case, that $\mathsf{seq}\, even \in \mathbb{P}(\mathsf{seq}\,\mathbb{Z})$. This is indeed true, and it is true in consequence of the monotonicity of seq: because $even \subseteq \mathbb{Z}$, we can deduce that $\mathsf{seq}\, even \subseteq \mathsf{seq}\,\mathbb{Z}$, in other words, that $\mathsf{seq}\, even \in \mathbb{P}(\mathsf{seq}\,\mathbb{Z})$.

The desire for monotonicity is what led to the definition of the set of bags over X in the standard Z library as the set of partial functions from X to the set $\mathbb{N}_1$ of strictly positive integers, rather than as the set of total functions from X to the natural numbers $\mathbb{N}$:

$$\mathsf{bag}\, X == X \pfun \mathbb{N}_1$$

Even apart from the enriched type system, the alternative definition of $\mathsf{bag}\, X$ as $X \fun \mathbb{N}$ has the disturbing property that the empty bag of even numbers $(\lambda x : even \bullet 0)$ is different from the empty bag of numbers $(\lambda x : \mathbb{Z} \bullet 0)$: in particular, as functions they have different domains.

The requirement of monotonicity also means that $\fun$ itself cannot be a type abbreviation in our enriched type system; our type system will be able to record the information that a relation is a partial function (because we can make $\pfun$ into a type abbreviation), but it cannot record the information that a function is total. In addition to partial functions ($\pfun$), other generic set-valued constants in the standard Z library that satisfy the monotonicity requirement, and can

thus be named as type abbreviations, include finite sets ($\mathbb{F}$), finite functions ($\pfun$), sequences (seq), and bags (bag).

5. Type Inference with Abbreviations

Motivated by the examples given in the preceding section, we now present a type system for Z that uses type abbreviations. The type system depends on a specification that defines a number of generic constants marked as type abbreviations. With respect to a set of type abbreviations a, the set of *enriched types* is defined inductively as follows:

- If X is a basic type name of the specification, then X is a type.
- If t is a type, then $\mathbb{P}\, t$ is a type.
- If $t_1, \ldots, t_n$ are types, then $t_1 \times \cdots \times t_n$ is a type.
- If a is a type abbreviation and $u_1, \ldots, u_n$ are types, then $a[u_1, \ldots, u_n]$ is a type.

Full Z also has the schema product constructor mentioned earlier; we omit it here as adding nothing to the present discussion. Each type abbreviation a is associated with a generic type $\rho(a) = [z_1, \ldots, z_n] \bullet \mathbb{P}\, t$. In a well-formed type, the number of arguments of each application of a matches the number n of formal generic parameters of the type $\rho(a)$. The type t may contain further type abbreviations, but we assume that abbreviations are defined in a non-circular way: it must be possible to associate a *level* with each type abbreviation a in such a way that $\rho(a)$ contains only abbreviations of a lower level than a. In what follows, we abbreviate the list $u_1, \ldots, u_n$ by $\vec{u}$, and so on, so that a typical abbreviated type is written $a[\vec{u}]$, where a has a generic type $[\vec{z}] \bullet \mathbb{P}\, t$. We define an *official* type to be one that contains no type abbreviations.

The relationship between a type abbreviation and the type it abbreviates is defined by a relation $\sqsubseteq$ of *containment* that is defined by a set of inference rules, independent of those we shall later use to define the typing relation $::$. For clarity, we emphasize that the relation $\sqsubseteq$ holds between the *syntactic expressions* that are types t; it is related to the subset relation $\subseteq$ on the sets $|t|$ of elements of a type t, in that $t \sqsubseteq t'$ implies $|t| \subseteq |t'|$, but the two relations are different, just as the typing relation $::$ between syntactic expressions is different from the membership relation $\in$ between mathematical objects.

The first inference rule says that each type abbreviation is contained in the type that it abbreviates, so that (for example) $\mathsf{seq}\, t \sqsubseteq \mathbb{N} \pfun t$ for each type t:

$$\frac{}{a[\vec{u}] \sqsubseteq t\{\vec{u}/\vec{z}\}} \quad [\, \rho(a) = [\vec{z}] \bullet \mathbb{P}\, t \,]$$

Second, the relation $\sqsubseteq$ is reflexive and transitive, as reflected in the following rules:

$$\frac{}{t \sqsubseteq t}$$

$$\frac{t_1 \sqsubseteq t_2 \qquad t_2 \sqsubseteq t_3}{t_1 \sqsubseteq t_3}$$

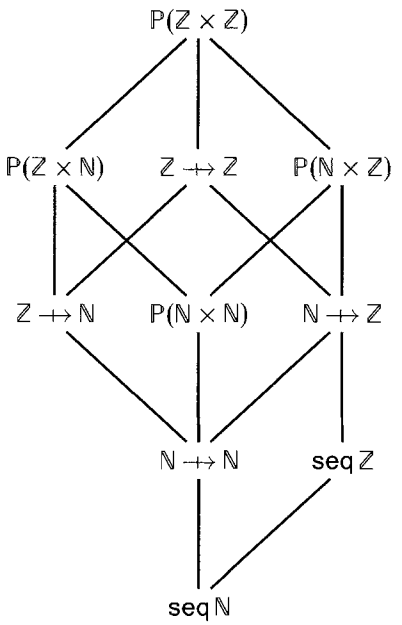

Fig. 1. Containment relation.

Third, since each type abbreviation a is monotonic with respect to inclusion, it respects the relation $\sqsubseteq$:

$$\frac{u_i \sqsubseteq u_i' \quad (1 \leq i \leq n)}{a[\vec{u}] \sqsubseteq a[\vec{u}']}$$

As an example of this rule, since $\mathbb{N} \sqsubseteq \mathbb{Z}$, we can conclude that $\mathsf{seq}\,\mathbb{N} \sqsubseteq \mathsf{seq}\,\mathbb{Z}$. Analogous rules of monotonicity apply to the basic type constructors $\mathbb{P}$ and $\times$:

$$\frac{t \sqsubseteq t'}{\mathbb{P}\, t \sqsubseteq \mathbb{P}\, t'}$$

$$\frac{t_1 \sqsubseteq t_1' \quad \ldots \quad t_n \sqsubseteq t_n'}{t_1 \times \cdots \times t_n \sqsubseteq t_1' \times \cdots \times t_n'}$$

These rules defining the containment relation $\sqsubseteq$ can give rise to elaborate networks of relationships among types. For example, given the three abbreviations

$$\mathbb{N} \sqsubseteq \mathbb{Z}, \quad X \leftrightarrow Y \sqsubseteq \mathbb{P}(X \times Y), \quad \mathsf{seq}\, X \sqsubseteq \mathbb{N} \leftrightarrow X$$

the enriched type $\mathsf{seq}\,\mathbb{N}$ has the network of containing types shown in Fig. 1.

The rest of this section is organized as follows: we first explain how the system of typing rules from Section 2 can be modified to assign enriched types to expressions. This modified system has the property that an expression E may be assigned many different types that differ in the extent to which type abbreviations have been expanded. We then state a number of results about the containment relation $\sqsubseteq$ that allow us to prove that each expression has a least type, if it has

any type at all. This proof is presented in the form of a new set of typing rules that assigns to each well-typed expression its unique least type.

We begin, therefore, by changing our system of typing rules so that it assigns enriched types in place of the previous official types. The first change is that the environment ρ now assigns enriched types to the variables of a specification, and not simply official types. The second change is that we add a new inference rule of *weakening* that allows type abbreviations to be expanded at any point in a derivation:

$$\frac{E :: t}{E :: t'} \quad [\, t \sqsubseteq t' \,]$$

Because of this rule, we can say that if an expression E is assigned type t by the rules, then it is also assigned all types t' such that $t \sqsubseteq t'$. Part of the rule's importance is that it allows types to be expanded where this is necessary for subsequent rules to be applicable.

The third change is that a special rule may be used to assign types to generic constants that have been named as type abbreviations. If a is a type abbreviation with n parameters, then the following rule may be used:

$$\frac{E_i :: \mathbb{P}\, u_i \quad (1 \le i \le n)}{a[\vec{E}] :: \mathbb{P}(a[\vec{u}])}$$

This rule is especially important when it is applied to the right hand side of a declaration $x : a[\vec{E}]$, because it then allows the identifier x to be added to the environment with the abbreviated type $a[\vec{u}]$, and later to be assigned that type when it is used in expressions.

The rest of the typing rules require no change, but if some of the generic constants from the Z library have been marked as type abbreviations, then some of the rules can be tightened up to give more accurate types. The existing rule for a sequence display $\langle E_1, \ldots, E_n \rangle$ gives it the type $\mathbb{P}(\mathbb{Z} \times t)$ in the official type system, where t is the type shared by all of $E_1, \ldots, E_n$. In the extended type system, if seq is a type abbreviation, then we can use a rule for sequence displays that gives the abbreviated type $\mathsf{seq}\, t$:

$$\frac{E_1 :: t \quad \ldots \quad E_n :: t}{\langle E_1, \ldots, E_n \rangle :: \mathsf{seq}\, t}$$

A similar improvement is possible in the rule for bag displays.

We now state three basic results about the relation $\sqsubseteq$. The proof of these results is given in an appendix.

1. For each type t there is a unique official type $\hat{t}$ such that $t \sqsubseteq \hat{t}$.
2. If $\hat{t}_1 = \hat{t}_2$ then t_1 and t_2 have a least upper bound $t_1 \sqcup t_2$. Standard results of abstract algebra guarantee that the operation $\sqcup$ is idempotent, commutative and associative, because $\sqsubseteq$ is a partial order.[2]
3. If $\hat{t} = t_1 \times \cdots \times t_n$ then there are least types $u_1, \ldots, u_n$ such that $t \sqsubseteq u_1 \times \cdots \times u_n$. Similarly, if $\hat{t} = \mathbb{P}\, t_1$ then there is a least type u_1 such that $t \sqsubseteq \mathbb{P}\, u_1$.

[2] Note that the type $t_1 \sqcup t_2$ is not necessarily equal as a set to $t_1 \cup t_2$. For example, with $\mathbb{N} \sqsubseteq \mathbb{Z}$ we have $(\mathbb{N} \times \mathbb{Z}) \sqcup (\mathbb{Z} \times \mathbb{N}) = \mathbb{Z} \times \mathbb{Z}$.

The first result immediately allows us to show that an expression is well-typed according to the modified rules exactly if it is well-typed according to the original rules given in Section 2. Let us write $::_0$ for the relation between expressions and official types defined by the original rules, and $::$ for the relation between expressions and enriched types defined by the modified rules. The two typing relations are linked by the following properties:

- $\rho \vdash E :: t$ implies $\hat{\rho} \vdash E ::_0 \hat{t}$.
- $\hat{\rho} \vdash E ::_0 t$ implies $\rho \vdash E :: t$.

We use the notation $\rho \vdash E :: t$ to denote the fact that the typing $E :: t$ is derivable with respect to the environment ρ, and we write $\hat{\rho}$ for the official environment such that for all x, if $\rho(x) = t$ then $\hat{\rho}(x) = \hat{t}$.

Any derivation of $E :: t$ can be converted into a derivation of $E ::_0 \hat{t}$ by replacing every type u that appears in the derivation by the official type $\hat{u}$ and deleting any weakening steps, which become trivial. Conversely, every derivation of $E ::_0 t$ for an official type t can be converted to a derivation of $E :: t$ by inserting a weakening step just after each leaf of the derivation tree, using the fact that $u \sqsubseteq \hat{u}$ to expand all type abbreviations immediately.

The second and third results allow us to show that each well-typed expression has a unique least type[3]. We shall do so by giving a third set of inference rules that define a relation $::'$ between expressions and enriched types. This relation has the following properties:

- For each expression E, there is at most one type t such that $E ::' t$.
- If $E ::' t$ then $E :: t$.
- If $E :: t$ then there exists a type t' such that $E ::' t'$ and $t' \sqsubseteq t$.

Again, many of the typing rules from the official type system can be used unchanged: for example, the existing rule for ordered pairs (E_1, E_2) allows arbitrary types for E_1 and E_2, and the type of the whole expression is assembled from the types of its elements. This means that the least type of (E_1, E_2) can similarly be assembled from the least types of E_1 and E_2:

$$\frac{E_1 ::' t_1 \quad \dots \quad E_n ::' t_n}{(E_1, \dots, E_n) ::' t_1 \times \cdots \times t_n}$$

Some existing rules require sub-expressions to have types with a certain form. The third result allows us to translate these rules systematically, by matching the least types of the sub-expressions with 'patterns' that describe the required form. We shall use the informal notation $t \hookrightarrow u_1 \times u_2$ to denote the fact that $\hat{t}$ has the form $t_1 \times t_2$, and u_1 and u_2 are the least types such that $t \sqsubseteq u_1 \times u_2$. Repeated applications of the result allow us to use nested patterns, as in the notation $t \hookrightarrow \mathbb{P}(u_1 \times u_2)$.

Using this notation, here is the rule for Cartesian product:

$$\frac{E_1 ::' t_1 \quad \dots \quad E_n ::' t_n}{E_1 \times \cdots \times E_n ::' \mathbb{P}(u_1 \times \cdots \times u_n)} \quad [\, t_1 \hookrightarrow \mathbb{P}\, u_1, \dots, t_n \hookrightarrow \mathbb{P}\, u_n \,]$$

[3] Here we again ignore the empty set display and similar expressions.

In this rule, the types u_i are obtained by matching the types t_i ascribed to E_i against the pattern $\mathbb{P}\, u_i$, expanding type abbreviations only as far as necessary to make the pattern match.

Other existing rules require the types of several sub-expressions to be the same. An example is the rule for set displays, which requires all the element expressions to have the same type. In translating these rules into rules for the least typing relation $::'$, we must change them to allow for expansion. Here is the translated rule for set displays:

$$\frac{E_1 ::' t_1 \quad \ldots \quad E_n ::' t_n}{\{E_1, \ldots, E_n\} ::' \mathbb{P}(t_1 \sqcup \cdots \sqcup t_n)} \quad [\, \hat{t}_1 = \hat{t}_2 = \cdots = \hat{t}_n \,]$$

For the expression $\{E_1, \ldots, E_n\}$ to be well-typed, all the expressions E_i must have the same official type, but they need not have the same enriched type. Even if this condition fails, enriched types can still be used by a type checking program in reporting the error, since two enriched types with different expansions are necessarily disjoint. The type of the whole expression is formed from the union of the types of the elements.

As a final example, we translate the typing rule for application.

$$\frac{E_1 ::' t_1 \qquad E_2 ::' t_2}{E_1(E_2) ::' u_2} \quad [\, t_1 \hookrightarrow \mathbb{P}(u_1 \times u_2); \hat{u}_1 = \hat{t}_2 \,]$$

This rule combines features of the rules for Cartesian products and set displays; the type of E_1 must expand to give a type of the form $\mathbb{P}(u_1 \times u_2)$, and the official expansion of u_1 must be the same as the official expansion of the type t_2 ascribed to E_2. Whenever the expression is defined, it has a value that lies in the type u_2, so that is the type ascribed to the whole expression.

We have now introduced three sets of typing rules for expressions, and have established relationships between the first and second sets and between the second and third sets. The first set of rules expresses the official type system, and the third set is the one we intend to implement in a type-checker. It is therefore worthwhile to summarize the relationship we have established between the first and third sets of rules. Both rules have the property that they assign unique types to expressions, and the following *correspondence principle* holds:

> For each expression E and environment ρ, an official type t_0 satisfies $\hat{\rho} \vdash E ::_0 t_0$ if and only if there is an enriched type t such that $\rho \vdash E ::' t$ and $t_0 = \hat{t}$.

This principle guarantees that an expression is recognized as well-typed using the new rules exactly if it is well-typed according to the official rules, and the type assigned to it by the new rules is consistent with that assigned to it by the official rules.

Like the official typing rules, the rules defining $::'$ can be extended to allow generic constants with implicit parameters. The typing rule is as follows:

$$\frac{}{x ::' t\{\vec{u}/\vec{z}\}} \quad [\, \rho(x) = [\vec{z}] \bullet t\,; u_1, \ldots, u_n \text{ official} \,]$$

This rule allows any *official* types to be substituted for the formal parameters of a generic constant, in accordance with the language rule that implicit parameters are always (official) types. As before, we say that an expression is well-typed if there is

a unique way of deriving its type using this rule for implicit parameters. A bottom-up type checker can be implemented using type variables as place-markers for unknown types, and using a unification algorithm that binds unknown variables to official types, fully expanding any type abbreviations first. We explore in the next section the consequences of this rule, and the extent to which it can be relaxed in particular cases, allowing some type abbreviations to be left unexpanded.

6. Tame Functions

Consider the expression $rev(\langle 1, 2, 3\rangle)$, in which the generic function *rev* is used to reverse a sequence of natural numbers. With $\mathbb{N}$ and $\mathsf{seq}\, X$ as type abbreviations, our typing algorithm has no difficulty in assigning to the argument $\langle 1, 2, 3\rangle$ the type $\mathsf{seq}\,\mathbb{N}$ in place of the official type $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$. What about the whole expression? The symbol *rev* is generic, and it is used here with implicit parameters, so our type checking algorithm invents a type variable α to stand for its generic parameter:

$$rev :: \mathsf{seq}\,\alpha \pfun \mathsf{seq}\,\alpha$$

What should be substituted for the type variable α? It is tempting to suppose that the answer is $\mathbb{N}$, but that is not correct, because of the language rule that implicit parameters are chosen as the *types* that can be deduced from the context, and the official type here is $\mathbb{Z}$, not $\mathbb{N}$. Thus we deduce the value $\mathbb{Z}$ for the type variable α. This instance of *rev* therefore has type $\mathsf{seq}\,\mathbb{Z} \pfun \mathsf{seq}\,\mathbb{Z}$, and after expansion, the type of the actual argument $\langle 1, 2, 3\rangle$ matches the domain type $\mathsf{seq}\,\mathbb{Z}$. The type of the whole expression is therefore deduced as $\mathsf{seq}\,\mathbb{Z}$.

Although this type is correct, it is a little disappointing because we know that the actual value of the expression, $\langle 3, 2, 1\rangle$, is a sequence over $\mathbb{N}$, and no harm would have been done if we had taken the value of α to be $\mathbb{N}$ instead of $\mathbb{Z}$. But this observation is based on a special property of *rev* that does not hold in general. In this example, the difference in information content between the types $\mathsf{seq}\,\mathbb{N}$ and $\mathsf{seq}\,\mathbb{Z}$ may seem small, but what is at stake here is the principle that *all* type abbreviations must be expanded when an implicit parameter is filled in from the context, and an unbounded amount of information may be thrown away in that expansion.

To avoid complete expansion, it is necessary to formalize the property of *rev* that guarantees that the result is in $\mathsf{seq}\,\mathbb{N}$ rather than $\mathsf{seq}\,\mathbb{Z}$. We call this property *tameness*, and give a formal definition later; in a sense, it expresses the property that the behaviour of a generic function does not depend on its generic parameter, but only the value of its argument. It is simplest to explain the concept of tameness first in the case of a total function like *rev* that has a single generic parameter, before giving the full definition. The tameness of *rev* is equivalent to the following:

$$\text{If } X' \subseteq X \text{ and } s \in \mathsf{seq}\, X', \text{ then } rev[X'](s) = rev[X](s)$$

Most of the generic functions defined in the standard Z library are tame, but a few are not. One of the latter is the generalized intersection operator

$$\bigcap[X] : \mathbb{P}(\mathbb{P}\, X) \longrightarrow \mathbb{P}\, X$$

This fails to be tame because of the way it treats the empty set, which depends on the parameter X: $\bigcap[X]\, \varnothing = X$. Another function in the standard library that is not tame is the reflexive–transitive closure operator $(\_^*)$. It is not tame because,

for any relation R, the reflexive–transitive closure $R^* = (\_^*)[X](R)$ contains the identity relation $\mathrm{id}\, X$, and so the value of this expression depends on the generic parameter X.

In general, a generic function f with type

$$[X_1, \ldots, X_n] \bullet t_1[X_1, \ldots, X_n] \pfun t_2[X_1, \ldots, X_n]$$

is said to be tame if the following property holds:

$$\text{If } X_i' \subseteq X_i \text{ for } 1 \leq i \leq n, \text{ then } t_1[\vec{X}'] \lhd f[\vec{X}] = f[\vec{X}']$$

In this definition, the notation $S \lhd f$ is used for the restriction of a function f to the set S. If $f : X \pfun Y$, then

$$S \lhd f = \{ x : X; y : Y \mid x \in S \land (x \mapsto y) \in f \bullet x \mapsto y \}$$

This definition also covers functions with multiple arguments, because of the Z convention that identifies them with functions whose single argument is a tuple.

Functions that are known to be tame can be treated specially when inferring types, by using the following rule:

$$\frac{E :' t_1\{\vec{u}/\vec{z}\}}{f(E) :' t_2\{\vec{u}/\vec{z}\}} \quad [\, \rho(f) \hookrightarrow [\vec{z}] \bullet \mathbb{P}(t_1 \times t_2); f \text{ tame} \,]$$

The expression $f(E)$ could also be treated by using the rules for generic constants and function application, but these would result in the extra condition that the types $u_1, \ldots, u_n$ should be official. The new rule avoids this condition, and thus can be applied without expanding abbreviations.

In a type checking program, the rule for tame functions can be implemented by marking the type variables introduced for the generic parameters of f as belonging to a tame function. During unification, such variables can be bound to enriched types without expanding them first.

7. Applications and Conclusion

The author's type checking program *fuzz* [Spi92b] allows generic constants to be marked either as type abbreviations or as tame functions. The standard library that the type checker loads before analysing the user's specification is already annotated in this way, and users are free to add their own annotations for extensions that they add to the standard library.

For the type checker to work properly, it is necessary that the generic constants that are marked as type abbreviations have a power-set type and are monotonic, and that those marked as tame functions have a function type and are genuinely tame. In each case, the first of the two stated requirements is a matter of typing, and the *fuzz* program can check that it is satisfied; but verifying the second requirement is more difficult, and the responsibility is left with the designer of the library. Errors here can affect the accuracy of the enriched types that the *fuzz* program displays for expressions, but not whether it reports errors in a specification, because although the types computed by the program are expressed in the enriched type system proposed in this paper, types are checked for equality essentially by expanding them into the official types first.

The strategy of displaying abbreviated types wherever possible is highly effective in improving the quality of error messages. For example, if s has been

declared as a sequence, then the predicate $s = \{0\}$ is ill-typed, and *fuzz* displays the following error message:

```
"example.tex", line 4: Types do not agree in equation
> Predicate: s = {0}
> LHS type:  seq NN
> RHS type:  P NN
```

Without type abbreviations, the types would have been displayed as $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$ and $\mathbb{P}\,\mathbb{Z}$, making the error message slightly less clear. In more complex examples, the increase in clarity is more dramatic. The expression $(\_ \frown \_) \circ (\_ \frown \_)$ composes the concatenation operator $\frown$ with itself. It is ill-typed because the operator expects a *pair* of sequences, but it returns a single sequence. *Fuzz* displays the following error message:

```
"example.tex", line 8: Right argument of operator \circ has wrong type
> Expression: (_ \cat _) \circ (_ \cat _)
> Arg type:   seq ? x seq ? -+> seq ?
> Expected:   ? <-> seq ? x seq ?
```

This message clearly shows that the expression $(\_ \frown \_)$ is a function from pairs of sequences to sequences; the type of the elements has not so far been determined, and is displayed as a question mark. The composition operator $\circ$ expects relations as its arguments, and it expects the target type of the right-hand argument to match the source type of the left-hand argument, which plainly does not happen in the example. Without type abbreviations, the types in this example would be displayed as $\mathbb{P}((\mathbb{P}(\mathbb{Z} \times ?) \times \mathbb{P}(\mathbb{Z} \times ?)) \times \mathbb{P}(\mathbb{Z} \times ?))$ and $\mathbb{P}(? \times (\mathbb{P}(\mathbb{Z} \times ?) \times \mathbb{P}(\mathbb{Z} \times ?)))$. The complexity of these expressions speaks for itself.

The type system proposed in this paper does not *replace* the 'official' type system based on $\mathbb{P}$ and $\times$; specifications are still well-typed exactly when the official type system succeeds in ascribing a unique type to every expression. Instead, the richer type system proposed here is to be used *in addition to* the official type system, as a way of extracting further information from a specification in an organized way. As we have seen, this extra information can be used to produce more helpful error messages in a type checking program.

Other possible applications of the richer type system are to generate indexes of the specification that give more helpful typing information for identifiers, and to support interactive browsing of a specification or machine-assisted proof of theorems about it. In machine-assisted proof, whether it is carried out fully automatically, or with the machine performing the house-keeping tasks of checking a human-directed proof, one of the most persistent problems is dealing with all the trivial conditions that must be checked. In reasoning about Z specifications, many of these conditions are assertions that values belong to particular sets, when their membership of these sets is 'obvious' from the 'types' of the expressions involved. The author hopes that, by formalizing and automating the calculation of these types, the work reported in this paper will contribute to making automated reasoning more effective.

The theorem prover implemented by Boyer and Moore [BoM79] makes similar use of heuristically-derived types, although their type system is limited to a finite number of disjoint classes of objects, and they do not have type constructors that lead to complex types with a nested structure. They report, however, that even their very simple type system contributes significantly to the efficiency of their theorem prover in dealing with 'obvious' conjectures.

The idea of a type system that supports inclusion relations among types has been exploited by Cardelli [Car85] and others in the study of type systems for object-oriented programming languages, and by Mitchell [Mit90] in the study of second-order lambda calculus. The purpose and formulation of the type system proposed here differs significantly from these sources, however. More closely related is the 'order-sorted' algebra used in the specification language OBJ [GoM92]. Unlike OBJ, the type system proposed here includes type constructors, and allows for generic objects in addition to generic theories.

Acknowledgements

The author is grateful to Stephen King for his perceptive comments on content and presentation, and to an anonymous reviewer for many suggestions for improvement.

The *fuzz* type checker described in the paper may be obtained from The Spivey Partnership, 10 Warneford Road, Oxford OX4 1LU, UK.

References

[BoM79] Boyer, R. and Moore, J. S.: *A computational logic*. Academic Press, 1979.

[Car85] Cardelli, L.: 'A Semantics of Multiple Inheritance', in *Semantics of Data Types*, (G. Kahn, D. B. MacQueen and G. Plotkin, eds.), LNCS 173, Springer-Verlag, 1985

[Der82] Dershowitz, N.: 'Orderings for term-rewriting systems', *Theoretical Computer Science*, March 1982.

[GoM92] Goguen, J. A. and Meseguer, J.: 'Order-sorted algebra (I): Equational deduction for multiple inheritance, overloading, exceptions and partial operations', *Theoretical Computer Science*, 105, 2 (1992), pp. 217–73.

[Mit90] Mitchell, J. C.: 'Polymorphic type inference and containment', in *Logical foundations of functional programming* (G. Huet, ed.), Addison–Wesley, 1990.

[Nic95] Nicholls, J. E. (ed.): *Draft Z standard*, Oxford University Computing Laboratory, 1995.

[Mil78] Milner, A. J. R. G.: 'A theory of type polymorphism in programming languages', *Journal of Computer and System Science*, 17 (1978), pp. 348–57.

[Spi88] Spivey, J. M.: *Understanding Z: a specification language and its formal semantics*, Cambridge University Press, March 1988.

[Spi92a] Spivey, J. M.: *The Z notation: a reference manual*, Prentice–Hall International, November 1989. Second edition, March 1992.

[Spi92b] Spivey, J. M.: *The fuzz manual*, The Spivey Partnership, 1992.

[SpS90] Spivey, J. M. and Sufrin, B. A.: 'Type inference in Z', in *VDM and Z – Formal methods in software development*, (D. Bjørner, C. A. R. Hoare and H. Langmaack, eds.), LNCS 428, Springer-Verlag, April 1990.

Appendix: Proof of Results

In this appendix, we give the proofs of a number of results about the containment relation $\sqsubseteq$ between types, and principally the three results stated in Section 5. The author has verified these results using the Boyer-Moore theorem prover [BoM79] for the special case where every type abbreviation has two arguments.

We begin by proving termination of the process of expanding type abbreviations. This is equivalent to proving termination of a system of rewrite rules that contains one rule $a[\vec{z}] \longrightarrow t$ for each type abbreviation a with $\rho(a) = [\vec{z}] \bullet \mathbb{P}\, t$. A step of rewriting with this rule amounts to replacing an an occurrence $a[\vec{u}]$ of the type abbreviation a with the instance $t\{\vec{u}/\vec{z}\}$ of the right hand side. We have said

that it must be possible to assign to each type abbreviation a a natural number $level(a)$ such that the expansion of each a contains only abbreviations of a lower level than a. This makes termination of the expansion process obvious, since it amounts to no more than non-recursive macro expansion. Nevertheless, we go to the trouble of giving a termination ordering, because later proofs will use induction on this ordering. One termination ordering that works is the recursive path ordering (see [Der82]) induced by the assignment of levels to type abbreviations. The reference cited includes proofs that this partial order is well-founded and respects the type constructors.

Alternatively, an explicit termination measure may be constructed as follows. If t is a type and X is a given set name, we define a measure $deg(t, X)$. Informally, $deg(t, X)$ is the number of occurrences of X in $\hat{t}$ – but since we have not yet proved the existence of $\hat{t}$, we must not use this as the formal definition. Instead, we use the following recursive definition:

$$deg(X, X) = 1$$

$$deg(Y, X) = 0 \quad (Y \neq X)$$

$$deg(\mathbb{P}\, t, X) = deg(t, X)$$

$$deg(t_1 \times \cdots \times t_n, X) = deg(t_1, X) + \cdots + deg(t_n, X)$$

$$deg(a[\vec{u}], X) = \sum_i deg(t, z_i) * deg(u_i, X) \text{ where } \rho(a) = [\vec{z}] \bullet \mathbb{P}\, t$$

This is a good definition because the measure $(level(t), size(t))$ goes down lexicographically in each recursive call, where by extension $level(t)$ for a type t is the maximum level of any type abbreviation occurring in t, and $size(t)$ is simply the number of symbols in t. We may assume because of the strictness of each type abbreviation a that $deg(t, z_i) > 0$ for each i, where $\rho(a) = [\vec{z}] \bullet \mathbb{P}\, t$.

Next, we define a measure $rank(t)$, again by recursion on $(level(t), size(t))$:

$$rank(X) = 0$$

$$rank(\mathbb{P}\, t) = rank(t)$$

$$rank(t_1 \times \cdots \times t_n) = rank(t_1) + \cdots + rank(t_n)$$

$$rank(a[\vec{u}]) = 1 + rank(t) + \sum_i deg(t, z_i) * rank(u_i)$$

$$\text{where } \rho(a) = [\vec{z}] \bullet \mathbb{P}\, t$$

The measure $rank(t)$ provides a termination ordering for expansion of type abbreviations. Informally, $rank(t)$ is the maximum number of steps of type expansion that t can undergo before it becomes official. It is easy to show by structural induction on t that

$$rank(t\{\vec{u}/\vec{z}\}) = rank(t) + \sum_i deg(t, z_i) * rank(u_i)$$

The only difficult case in this proof arises when $t = a[\vec{v}]$ and the types v_i satisfy the equation. In this case, if $\rho(a) = [\vec{x}] \bullet \mathbb{P}\, w$, then

$$\begin{aligned} &rank(a[\vec{v}]\{\vec{u}/\vec{z}\}) \\ &\quad = rank(a[\vec{v}\{\vec{u}/\vec{z}\}]) \\ &\quad = 1 + rank(w) + \sum_j deg(w, x_j) * rank(v_j\{\vec{u}/\vec{z}\}) \\ &\quad = 1 + rank(w) + \sum_j deg(w, x_j) * \left(rank(v_j) + \sum_i deg(v_j, z_i) * rank(u_i)\right) \end{aligned}$$

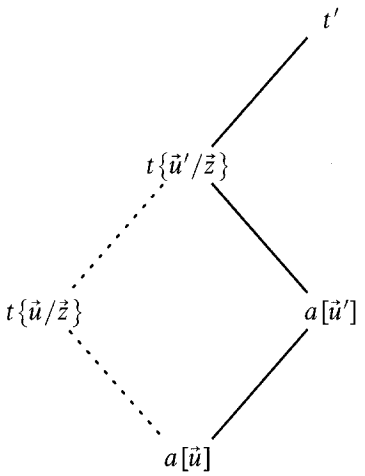

Fig. 2. Expansion lemma.

$$
\begin{aligned}
&= 1 + rank(w) + \textstyle\sum_j deg(w, x_j) * rank(v_j) \\
&\quad + \textstyle\sum_i \left(\sum_j deg(w, x_j) * deg(v_j, z_i) \right) * rank(u_i) \\
&= rank(a[\vec{v}]) + \textstyle\sum_i deg(a[\vec{v}], z_i) * rank(u_i)
\end{aligned}
$$

Thus if a is any type abbreviation, then $rank(a[\vec{u}]) = rank(t\{\vec{u}/\vec{z}\}) + 1$ where $\rho(a) = [\vec{z}] \bullet \mathbb{P}\, t$, and the rank goes down by one when a type abbreviation is expanded directly. Because of strictness, each type constructor is strictly monotonic with respect to the *rank* measure, so this measure decreases when a type abbreviation is expanded anywhere inside a type, and expansion of type abbreviations therefore terminates after a finite number of steps. We shall often use induction on the compound measure $(rank(t), size(t))$ under the lexicographic ordering; we call this *type induction.*

We now prove the following *expansion lemma*, which means that every expansion of $a[\vec{u}]$ can be obtained either simply by expanding types from $\vec{u}$, or by expanding the abbreviation a as the very first step, then expanding further the type that results from that step:

Proposition 1. If a is a type abbreviation, $\rho(a) = [\vec{z}] \bullet \mathbb{P}\, t$, and $a[\vec{u}] \sqsubseteq t'$ then either $t' = a[\vec{u}']$ for some types $\vec{u}'$ with $\vec{u} \sqsubseteq \vec{u}'$, or $t\{\vec{u}/\vec{z}\} \sqsubseteq t'$.

Proof. Suppose that $a[\vec{u}] \sqsubseteq t'$ and t' is not of the form $a[\vec{u}']$ with $\vec{u} \sqsubseteq \vec{u}'$. According to the rules defining $\sqsubseteq$, there is then a chain of types

$$a[\vec{u}] \sqsubseteq a[\vec{u}^{(1)}] \sqsubseteq \ldots \sqsubseteq a[\vec{u}^{(k)}] \sqsubseteq t\{\vec{u}^{(k)}/\vec{z}\} \sqsubseteq t^{(1)} \sqsubseteq \ldots \sqsubseteq t^{(m)} = t'$$

in which each step arises from the expansion of a single occurrence of a type abbreviation, and $\vec{u} \sqsubseteq \vec{u}^{(1)} \sqsubseteq \ldots \sqsubseteq u^{(k)}$. Taking $\vec{u}' = \vec{u}^{(k)}$, we have

$$a[\vec{u}] \sqsubseteq a[\vec{u}'] \sqsubseteq t\{\vec{u}'/\vec{z}\} \sqsubseteq t'$$

with $\vec{u} \sqsubseteq \vec{u}'$ (see Fig. 2). If $t\{\vec{u}/\vec{z}\} \sqsubseteq t\{\vec{u}'/\vec{z}\}$, then the proposition follows; and this last formula is easily proved by type induction on t. □

We next turn to the three results stated in Section 5. We begin with the first of them:

Proposition 2. For each type t there is a unique official type $\hat{t}$ such that $t \sqsubseteq \hat{t}$.

Proof. Type induction on t. If t is a basic type X, then t is already official, and we may take $\hat{t} = t$; this choice is plainly the only one possible. If t is a power set type $\mathbb{P} u$, where there is a unique official type $\hat{u}$ with $u \sqsubseteq \hat{u}$, then we take $\hat{t} = \mathbb{P} \hat{u}$. Plainly $t \sqsubseteq \hat{t}$; for its uniqueness, observe that if t' is such that $t = \mathbb{P} u \sqsubseteq t'$, then $t' = \mathbb{P} u'$ for some u' with $u \sqsubseteq u'$. If t' and hence u' is official, it follows that $u' = \hat{u}$, and so $t' = \hat{t}$. The case that t is a Cartesian product type $u_1 \times \cdots \times u_n$ is similar.

Finally, if t is an abbreviated type $a[\vec{u}]$ where $\rho(a) = [\vec{z}] \bullet \mathbb{P} w$, then $t \sqsubseteq t'$, where $t' = w\{\vec{u}/\vec{z}\}$, and t' has smaller rank than t. We may assume the existence of a unique official type $\hat{t}'$ such that $t' \sqsubseteq \hat{t}'$, and immediately $t \sqsubseteq t' \sqsubseteq \hat{t}'$, so we take $\hat{t} = \hat{t}'$. It remains to show that $\hat{t}'$ is the only official type t'' with $t \sqsubseteq t''$; for this we use the expansion lemma. Suppose $t \sqsubseteq t''$ and t'' is official. Since $a[\vec{u}] = t \sqsubseteq t''$, either $\hat{t} = a[\vec{u}']$ for some $\vec{u}'$, or $t' = w\{\vec{u}/\vec{z}\} \sqsubseteq t''$. Because t'' is official the first of these is impossible, so $t' \sqsubseteq t''$ and by uniqueness of $\hat{t}'$ we have $\hat{t} = \hat{t}' = t''$. □

Before proving the second result from Section 5, we state and prove a lemma that is derived from strictness.

Proposition 3. If a is a type abbreviation, and two types $t = a[\vec{u}]$ and $t' = a[\vec{u}']$ satisfy $\hat{t} = \hat{t}'$, then $\hat{u}_i = \hat{u}'_i$ for each i.

Proof. We shall prove the following by type induction on w: if $v = w\{\vec{u}/\vec{z}\}$ and $v' = w\{\vec{u}'/\vec{z}\}$ satisfy $\hat{v} = \hat{v}'$, and w contains an occurrence of z_i for some i, then $\hat{u}_i = \hat{u}'_i$. The desired result follows on taking $w = a[\vec{z}]$.

If w is a given set, then in fact $w = z_i$ and $\hat{u}_i = \hat{v} = \hat{v}' = \hat{u}'_i$.

If w is a power set type $\mathbb{P} y$, then y contains an occurrence of z_i, and we may assume that the result holds when w is replaced by y. Also, if $r = y\{\vec{u}/\vec{z}\}$ and $r' = y\{\vec{u}'/\vec{z}\}$ then $\mathbb{P} \hat{r} = \hat{v} = \hat{v}' = \mathbb{P} \hat{r}'$, so $\hat{r} = \hat{r}'$. So the induction hypothesis gives that $\hat{u}_i = \hat{u}'_i$ as required.

If w is a Cartesian product type $y_1 \times \cdots \times y_n$, then at least one of the y_j contains an occurrence of z_i. The argument proceeds as for the case $w = \mathbb{P} y$.

Finally, suppose w is a type abbreviation $a[\vec{y}]$, where $\rho(a) = [\vec{x}] \bullet r$, and suppose that the result holds if w is replaced by any type that is smaller in our ordering. Then $v = a[\vec{p}]$ where $p_j = y_j\{\vec{u}/\vec{z}\}$ for each j. So $v \sqsubseteq s$, where $s = r\{\vec{p}/\vec{x}\}$. Similarly, $v' \sqsubseteq s'$, where s' and p'_j are defined analogously to s and p_j. Thus $\hat{s} = \hat{v} = \hat{v}' = \hat{s}'$, and r is a type with smaller rank than w, since $rank(w) = 1 + rank(r) + \sum_j deg(r, x_j) * rank(y_j)$. Also, one of the y_j contains an occurrence of z_i; and because a is strict, r also contains an occurrence of this y_j. We therefore deduce (by replacing w by r in the induction hypothesis) that $\hat{p}_j = \hat{p}'_j$ for this value of j. Next, observe that y_j has smaller size and no greater rank than w, so may deduce (replacing w by y_j in the induction hypothesis) that $\hat{u}_i = \hat{u}'_i$ as required. □

We next prove the result itself:

Proposition 4. If $\hat{t} = \hat{t}'$ then t and t' have a least upper bound $t \sqcup t'$.

Proof. Simultaneous type induction on t and t'. If neither t nor t' is a type

abbreviation, then the argument is straightforward. We consider explicitly the following cases:

(a) $t = a[\vec{u}]$ and $t' = a[\vec{u}']$ are instances of the same type abbreviation.
(b) $t = a[\vec{u}]$ and $t' = b[\vec{u}']$ with $a \neq b$.
(c) $t = a[\vec{u}]$, but t' is not a type abbreviation.
(d) $t' = b[\vec{u}']$, but t is not a type abbreviation.

In each case, we may assume that the result holds if either t or t' is replaced by a type of smaller rank.

For case (a), since $\hat{t} = \hat{t}'$, our lemma tells us that $\hat{u}_i = \hat{u}_i'$ for each i. Also, each u_i has smaller rank than t, and each u_i' has smaller rank than t', so the induction hypothesis gives a least upper bound $s_i = u_i \sqcup u_i'$ for each i. We claim that $r = a[\vec{s}]$ is a least upper bound of t_1 and t_2. Plainly $t \sqsubseteq r$ and $t' \sqsubseteq r$; now suppose r' is any upper bound of t and t'. By the expansion lemma (twice), either $r' = a[\vec{s}']$ with $\vec{u} \sqsubseteq \vec{s}'$ and $\vec{u}' \sqsubseteq \vec{s}'$, or $w\{\vec{u}/\vec{z}\} \sqsubseteq r'$ and $w\{\vec{u}'/\vec{z}\} \sqsubseteq r'$, where $\rho(a) = [\vec{z}] \bullet \mathbb{P}\, w$. In the first case, leastness of s_i gives $s_i \sqsubseteq s_i'$ for each i, and so $r \sqsubseteq r'$. In the second case, we claim that $w\{\vec{s}/\vec{z}\} \sqsubseteq r'$; then $r = a[\vec{s}] \sqsubseteq w\{\vec{s}/\vec{z}\} \sqsubseteq r'$ as required. This last claim is easily proved by another type induction on w.

For case (b), suppose the level of a is greater than or equal to that of b; the other case is symmetric. Any common expansion r of t and t' is not of the form $a[\vec{y}]$, so by the expansion lemma $t_1 = w\{\vec{u}/\vec{z}\} \sqsubseteq r$, where $\rho(a) = [\vec{z}] \bullet \mathbb{P}\, w$. The type t_1 has smaller rank than t, so we may deduce from the induction hypothesis that t_1 and t' have a least upper bound $t_1 \sqcup t'$, which is also a least upper bound for t and t'. Cases (c) and (d) are similar, and again we expand either t or t' as necessary before appealing to the induction hypothesis. □

We finally prove the third result from Section 5:

Proposition 5. If $\hat{t} = t_1 \times \cdots \times t_n$ then there are least types $u_1, \ldots, u_n$ such that $t \sqsubseteq u_1 \times \cdots \times u_n$.

Proof. The proof is again by type induction on t. Since $\hat{t}$ is a Cartesian product $t_1 \times \cdots \times t_n$, it is plain that t itself cannot be a given set or power-set type, and if t is a Cartesian product $u_1 \times \cdots \times u_k$, then $k = n$ and $u_1, \ldots, u_n$ are the required least types.

The remaining case is when t is a type abbreviation $a[\vec{y}]$ with $\rho(a) = [\vec{z}] \bullet \mathbb{P}\, w$. We may assume that the result holds of $t' = w\{\vec{y}/\vec{z}\}$, which has smaller rank than t and satisfies $\hat{t}' = \hat{t} = t_1 \times \cdots \times t_n$. So let $u_1, \ldots, u_n$ be least types such that $t' \sqsubseteq u_1 \times \cdots \times u_n = v$, say. These u_i are the required least types for t, because the expansion lemma guarantees that if $t \sqsubseteq u_1' \times \cdots \times u_n' = v'$, say, then either $v' = a[\vec{r}]$ for some r (which is impossible), or $t' \sqsubseteq v'$, in which case $\vec{u} \sqsubseteq \vec{u}'$ as required. The proof of the analogous result for power-set types is similar. □

Received August 1995
Accepted in revised form March 1996 by D. J. Cooke