The Journal of Supercomputing (2022) 78:2988–3028
https://doi.org/10.1007/s11227-021-03839-9

Making legacy Fortran code type safe through automated program transformation

Wim Vanderbauwhede[1]

Accepted: 21 April 2021 / Published online: 14 July 2021

Abstract
Fortran is still widely used in scientific computing, and a very large corpus of legacy as well as new code is written in FORTRAN 77. In general this code is not type safe, so that incorrect programs can compile without errors. In this paper, we present a formal approach to ensure type safety of legacy Fortran code through automated program transformation. The objective of this work is to reduce programming errors by guaranteeing type safety. We present the first rigorous analysis of the type safety of FORTRAN 77 and the novel program transformation and type checking algorithms required to convert FORTRAN 77 subroutines and functions into pure, side-effect free subroutines and functions in Fortran 90. We have implemented these algorithms in a source-to-source compiler which type checks and automatically transforms the legacy code. We show that the resulting code is type safe and that the pure, side-effect free and referentially transparent subroutines can readily be offloaded to accelerators.

Keywords Fortran · Type safety · Type system · Program transformation · Acceleration

1 Introduction

1.1 The enduring appeal of Fortran

The Fortran programming language has a long history. It was originally proposed by John Backus in 1957 for the purpose of facilitating scientific programming, and has since become widely adopted amongst scientists, and been shown to be an effective language for use in supercomputing. Even today, Fortran is still the dominant language in supercomputing.

✉ Wim Vanderbauwhede
wim.vanderbauwhede@glasgow.ac.uk

[1] School of Computing Science, University of Glasgow, Glasgow, UK

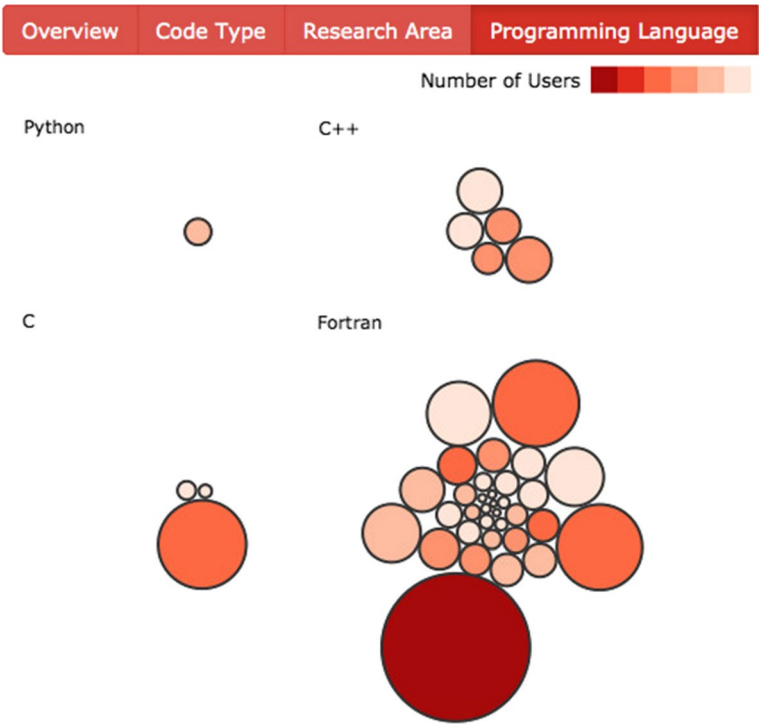

Fig. 1 The monthly usage of the UK Archer supercomputer per programming language (July 2016; more recent data not made available)

According to Yamamoto [1], 68% of the utilisation of the K computer (one of the largest supercomputers in the world) in 2014 was Fortran (using invocations of the compiler as a proxy). The monthly usage statistics of Archer, the largest supercomputer in the UK,[1] illustrated in Fig. 1 show an even higher ratio.

Fortran is still actively developed, and the most recent standard is Fortran 2018 (ISO/IEC 1539:2018), released in November 2018 [14]. However, adoption of recent standard is quite slow. Figure 2 shows the relative citations (citations per revision normalised to sum of citations for all revisions) for Google Scholar and ScienceDirect for each of the main revisions of Fortran. We collected results for the past 10 years (2006–2016) and also since the release of FORTRAN 77 (1978–2019). As an absolute reference, there were 15,100 citations in Google Scholar mentioning FORTRAN 77 between 2009 and 2019. It is clear that Fortran-77 is still widely used and that the latest standards (2003, 2008, 2018) have not yet found widespread adoption.

Based on the above evidence (confirmed by our own experience of collaboration with scientists), the current state of affairs is that for many scientists, FORTRAN 77

[1] http://www.archer.ac.uk/status/codes/.

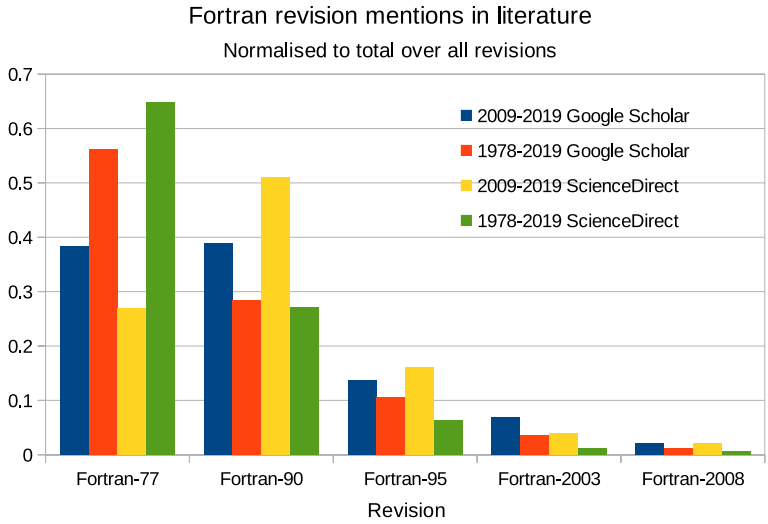

Fig. 2 Literature mentions of different revisions of Fortran using Google Scholar and ScienceDirect

is still effectively the language of choice for writing modells. Even if the code adopts Fortran 90 syntax, in practice very few of the semantic extensions are used, so that from a semantic perspective the code is FORTRAN 77. There is also a vast amount of legacy code in FORTRAN 77. Because the FORTRAN 77 language was designed with assumptions and requirements very different from today, code written in it has inherent issues with readability, scalability, maintainability and parallelisation. A comprehensive discussion of the issues can be found in [18]. As a result, many efforts have been aimed at refactoring legacy code, either interactive or automatic, and to address one or several of these issues. Our work is part of that effort, and we are specifically interested in automatically refactoring Fortran for offloading to accelerators such as GPUs and FPGAs.

1.2 Acceleration by offloading matters

Hardware accelerators have proven extremely effective in accelerating scientific code. Of the Green Top 10, eight systems use accelerators. However, in practice the accelerators have their own memory, and the most common compute modell is still to offload part of the calculation to the accelerator and copy the results back to the host memory. Even if the accelerator is cache-coherent with the host memory, having the code to be run on the accelerator in a separate memory space is still advantageous as it results in reduced coherency traffic.

1.3 The need for pure functions

Because of the separation of memory spaces and the absence of an operating system on the accelerator, the code units offloaded to the accelerator must be self-contained:

- No shared memory space (COMMON blocks)
- No system calls in general and no I/O operations in particular
- No library calls except intrinsic ones

A routine which meets these requirement is equivalent to a *pure function*: for a given set of input values, it always produces the same outputs values, and it only influences the rest of the world through these output values. Therefore, any other mechanism to share data (specifically COMMON blocks) is not allowed.

A kernel offloaded to an accelerator is in general expected to behave as a pure function: the inputs are the data copied to the accelerator's memory and the outputs the data copied from the accelerator's memory. Therefore, a key requirement for offloading code units to accelerators is that they are pure functions. Note that this implies "no I/O system calls" because these would cause the function to be impure. The restriction on library calls is a practical one because they can't be incorporated into the binary for the accelerator. From a "pure function" perspective, calls to library functions are acceptable if the library functions themselves are pure.

1.4 The case for type safety

1.4.1 What is type safety

In his paper, "A Theory of Type Polymorphism in Programming" [10], Robin Milner expressed the notion of type safety as "Well typed programs cannot go wrong." By "going wrong", we mean in general not computing the expected result. There are several components contributing to this behaviour: one is the language's type system, the other is the type checker, and finally there is the actual program code.

In a type-safe language, the language's type system ensures programs cannot perform operations that are not compatible with the types of the operands involved, i.e. there are no type errors in a well-typed program written in a type-safe language. By type error, we mean an error arising from the fact that a variable (or constant or function) with a given type is treated as if it has a different type.

A type checker is called *sound* if it only accepts correctly typed programs. However, the fact that a sound type checker accepts a correctly typed program does not mean the program is correct.

1.4.2 Type safety in Fortran

In the context of Fortran, the type system as specified in "ANSI X3.9-1978—American National Standard Programming Language FORTRAN" [2], hereafter called the

"f77 specification", is not type-safe. It is possible to write programs which the type checker accepts but are nonetheless incorrect from the perspective of the type system. The key culprit for this is the loss of type information which occurs when data are handled via COMMON blocks or EQUIVALENCE statements.

2 Related work

2.1 Formalisation of Fortran

There has been surprisingly little research into Fortran's type system. There is some work on formalisation of data abstraction, specifically encapsulation to create abstract arrays in FORTRAN 77 [3] and on the formal specification of abstract data types implemented through derived types in Fortran 90 [9, 16]. There is also some work on the formalisation of Fortran 95 semantics using VDM [15] but there is no publication on the final outcome. Specifically with regards to the type system, the only work that we are aware of is on the extension of Fortran 90 types with an attribute reflecting the unit of measurement [4]. According to our survey, a formalisation of the FORTRAN 77 type system or an analysis of its type safety has not been reported before.

2.2 Source-to-source compilation and refactoring

There are a number of source-to-source compilers and refactoring tools for Fortran available. However, very few of them actually support FORTRAN 77. The most well known are the ROSE framework[2] from LLNL [8], which relies on the Open Fortran Parser (OFP[3]). This parser claims to support the Fortran 2008 standard. Furthermore, there is the language-fortran[4] parser which claims to support FORTRAN 77 to Fortran 2003. A refactoring framework which claims to support FORTRAN 77 is CamFort [11], according to its documentation it supports Fortran 66, 77 and 90 with various legacy extensions. That is also the case for the Eclipse-based interactive refactoring tool Photran [12], which supports FORTRAN 77—2008. These tools are very useful, indeed both CamFort and Photran provide powerful refactorings. As we shall discuss in more detail below, for effective refactoring of common blocks, and determination of data movement direction, whole-source code (inter-procedural) analysis and refactoring is essential. A long-running project which does support inter-procedural analysis is PIPS,[5] started in the 1990's. The PIPS tool does support FORTRAN 77 but does not supported the refactorings, we propose. For completeness, we mention the commercial solutions plusFort[6] and VAST/77to90[7] which both

[2] http://www.rosecompiler.org/index.html.
[3] http://fortran-parser.sourceforge.net/.
[4] https://hackage.haskell.org/package/language-fortran.
[5] http://pips4u.org/.
[6] http://www.polyhedron.com/pf-plusfort0html.
[7] http://www.crescentbaysoftware.com/compilertech.html.

can refactor common blocks into modules but not into procedure arguments. In conclusion, there are many projects that provide refactoring program transformations. However, none of them focus on the type safety of the resulting programs.

3 Contribution

We present in this paper a formal analysis of the type safety of normalised FORTRAN 77 programs (Sect. 4) and a series of algorithms for program transformation into normalised form (Sect. 6). We also present additional type checks for COMMON blocks and EQUIVALENCE associations (Sect. 7), as a precondition to the program transformation.

These algorithms are implemented in our source-to-source compiler[8] which can automatically rewrite FORTRAN 77 programs into Fortran 90 so as to remove all COMMON and EQUIVALENCE statements, provide full referential transparency and ensure that all functions marked for offloading to accelerators are pure[19]. The conversion from FORTRAN 77 to Fortran 90 is necessary because we rely on Fortran 90 features (most notably INTENT and IMPLICIT NONE) for the improved type safety. It does not impact the performance of the program.

We further show that (with a small number of additional restrictions) the resulting code is type safe when type checked against the type system which we present and well typed programs adhering to these restrictions will not go wrong if they are accepted by the type checker. What this means is that if an original FORTRAN 77 program is accepted by the type checker of our source-to-source compiler then the Fortran 90 program which it generates can be type checked with an ordinary Fortran compiler[9] with all type-based warnings turned into errors, and the code will type check cleanly.

We have validated our source-to-source compiler against the NIST (US National Institute of Standards and Technology) FORTRAN 78 test suite[10] which aims to validate adherence to the ANSI X3.9-1978 (FORTRAN 77) standard. Furthermore, we tested the compiler on a simple 2-D shallow water modell and a simple Coriolis force modell from [7], on the NASA Chemical Equilibrium with Applications program,[11] used to calculate e.g. theoretical rocket performance[5] and on the Large Eddy Simulator for Urban Flows,[12] a high-resolution turbulent flow modell [17].

[8] https://github.com/wimvanderbauwhede/RefactorF4Acc.

[9] We have tested this with GNU Fortran 9.3.0 and PGI/Nvidia Fortran 20.11-0 and verified the behaviour of the Intel, SunSoft/Oracle and Fujitsu/Lahey compilers from their documentation.

[10] http://www.itl.nist.gov/div897/ctg/fortran_form.htm.

[11] https://www1.grc.nasa.gov/research-and-engineering/ceaweb/.

[12] https://github.com/wimvanderbauwhede/LES.

4 Formal analysis of the type safety of normalised Fortran programs

In this work, a *normalised* FORTRAN 77 program is a program that consists of pure functions (see Sect. 4.2 for a formal definition) and where all variables, parameters and functions are explicitly typed. We discuss in Sect. 6 how to achieve fully explicitly typing and under which conditions a procedure can be made pure.

4.1 Type systems concepts and notation

A *type* is a formal mechanism to provide information about the expected form of the result of a computation. More precisely, if e is an expression, a *typing* of e as e.g. an integer is an assertion that when e is evaluated, its value will be an integer. Such an assertion is called a typing judgment. For such a typing judgement to be meaningful, e must be well typed. For any expression this means that it must be internally consistent as well as consistent with its context, i.e. if the expression e contains free variables, they must be declared with the right type in the context of the code unit. We will use the term *type statement* (as used in the f77 specification) or *type declaration* (more common in type theory) for the statements that declare the type of a constant, variable or function.

We will use the standard notation for typing rules as used for example in [13], which can be summarised as:

- The assertion "the expression e has type τ" is written as $e : \tau$
- If an assertion must hold for a certain context, i.e. a set of expressions with declared types such as a code unit, the context is conventionally denoted as Γ, and the operator $\vdash$ (called "turnstile" in type theory) is used to write an assertion of the form "assuming a context Γ then the expression e has type τ" is written as $\Gamma \vdash e : \tau$.
- The double arrow ($\Rightarrow$) is used to put additional constraints on a type and is read as "these constraints must apply to the type for the type judgement to hold"
- The type of a function of a single argument is written as $f : \tau_{in} \rightarrow \tau_{out}$, and the function itself is written without parentheses, so $y = f\,x$ rather than $y = f(x)$.
- We will write the type declaration for a tuple (ordered set) of m expressions as $(e_1, \ldots, e_m) : (\tau_1, \ldots, \tau_m)$ or for brevity as $\mathbf{e_m} : \mathbf{T_m}$.

We deviate slightly from the terminology used in the f77 specification in favour of the more common terminology: we will refer to the symbolic name of a datum as a variable rather than a variable name, and we will refer to what the f77 specification calls a variable as a scalar. Thus, a variable can be a scalar or an array.

4.2 The definition of a pure function

If a function is pure, then it must return a least one datum as a result, because otherwise it means it did not compute anything and can be removed as dead code.

Furthermore, a pure function without input arguments is effectively a constant, so we can also assume that the there is at least a single input variable. Therefore, we can without loss of generality assume that a function takes as a single argument a tuple of expressions and returns a tuple of expressions.

Let Γ be the context of a given program, i.e. the set of all variables with their type that are declared in a code unit. Consider a function

$$\begin{aligned} f : \mathbf{T}_{\mathbf{in,k}} &\rightarrow \mathbf{T}_{\mathbf{out,m}} \\ \mathbf{x}_{\mathbf{out,m}} &= f\, \mathbf{x}_{\mathbf{in,k}} \end{aligned}$$

This function is *pure* iff

$$\forall \Gamma, \forall \mathbf{x}_{\mathbf{in,k}} \in \Gamma, \exists ! \, \mathbf{x}_{\mathbf{out,m}} \in \Gamma : \mathbf{x}_{\mathbf{out,m}} = f\, \mathbf{x}_{\mathbf{in,k}}$$

In words, for any given context Γ where $\mathbf{x}_{\mathbf{in,k}}$ and $\mathbf{x}_{\mathbf{out,m}}$ are declared, then if $\mathbf{x}_{\mathbf{in,k}}$ is a given set of argument values of the correct type, the function will always return the same values $\mathbf{x}_{\mathbf{out,m}}$ regardless of the rest of the content of Γ. Note that the fact that $\Gamma$is the same for the inputs$\mathbf{x}_{\mathbf{in,k}}$, and the results of the function call $\mathbf{x}_{\mathbf{out,m}}$ implies that the function does not modify Γ. We will see in Sect. 6 and following how any Fortran procedure can be transformed into a pure function.

4.3 Specification of FORTRAN 77 data types

According to §4.1 *Data Types* of the f77 specification,

> The six types of data are:
>
> 1. Integer
> 2. Real
> 3. Double precision
> 4. Complex
> 5. Logical
> 6. Character

The f77 specification discusses each of these types in terms of their *storage units*. According to §2.13 *Storage*:

> A storage unit is either a numeric storage unit or a character storage unit. An integer, real, or logical datum has one numeric storage unit in a storage sequence. A double precision or complex datum has two numeric storage units in a storage sequence. A character datum has one character storage unit in a storage sequence for each character in the datum. This standard does not specify a relationship between a numeric storage unit and a character storage unit. If a datum requires more than one storage unit in a storage sequence, those storage units are consecutive.

Thus

Table 1 Relationship between storage unit, kind and bytes for FORTRAN 77 types

Type	Size in bytes (kind)	#Storage units (numeric)
`integer`	4	1
`real`	4	1
`double precision`	8	2
`complex`	8	2
`logical`	4	1
	Size in bytes	#Storage units (character)
`character`	1	1

- An integer or real has one storage unit
- A *double precision* datum has two consecutive numeric storage units in a storage sequence (§4.5 *Double Precision Type*).
- A *complex* datum is a processor approximation to the value of a complex number. The representation of a complex datum is in the form of an ordered pair of real data. The first of the pair represents the real part of the complex datum, and the second represents the imaginary part. Each part has the same degree of approximation as for a real datum. A complex datum has two consecutive numeric storage units in a storage sequence; the first storage unit is the real part, and the second storage unit is the imaginary part (§4.6 *Complex Type*).

As quoted above, the f77 specification does not specify the size of a storage unit. However, the consensus amongst the major Fortran compilers[13] is as follows:

Various extensions exists such as `byte`, `double complex`, etc. Technically, the use of *kinds* in type statements (e.g. `integer*8`) is not part of the f77 specification. It is however widely used and supported by all current Fortran compilers, specifically the open source GNU Fortran compiler g77. In this paper, we effectively consider FORTRAN 77 to be defined by the f77 specification combined with the g77 extensions.[14]

We will treat the *kind* as the number of bytes of storage as in Table 1 and define the scalar types as *(Typename, Kind)* tuples. Moreover, we define a character storage unit as 1 byte. This allows us the simplify the types to integer, real, complex and logical, because we can define a double precision as (real,8) and a character as (integer,1). For the rest of the discussion, we will treat the character type as an integer with a kind of 1 and a character string as an array of characters. We will not discuss any special cases for characters because it would needlessly complicate the discussion without adding anything material in terms of the type system.

[13] GNU, PGI/Nvidia, Intel, SunSoft/Oracle, Lahey/Fujitsu.

[14] https://gcc.gnu.org/onlinedocs/gcc-3.4.6/g77/Language.html.

4.4 Formalising the FORTRAN 77 type system

With the conventions from Sect. 4.3 and the above assumptions, we can describe Fortran's type system using sets of entities. The formal definition is given in "Appendix 1".

The general form of a type τ in FORTRAN 77 is constructed from the following:

- Primitive types (scalars or arrays)
- Tuple types
- Function types
- *void*
- Type variables

Tuple types are ordered sets of types, denoted as $(\tau_1, \tau_2, \dots)$ and used to represent the type of the arguments of a function or subroutine.

Function types represent the entire type of a function declaration. So $\tau_1 \to \tau_2$ is the type of a function that takes an input of a given type τ_1 and returns a result of a given type τ_2.

The *void* type is used in the typing rules for subroutine calls and assignments, as these are statements that do not have a type. A subroutine declaration therefore has type $\tau \to void$.

Type variables are variables that can represent any type. They arise a result of the polymorphism of the arithmetic and relational operators, as well as of some intrinsic functions, discussed in Sect. 4.4.6.

To investigate the type safety of this type system, we need to consider the typing rules for

- Constants
- Scalar and array declarations and accesses
- Function and subroutine declarations and applications
- Assignments
- Expressions

Furthermore, we need to consider specifically how Fortran handles subtypes, which arise in the context of what the f77 specification calls *type conversion*, also commonly known as type coercion.

4.4.1 Constants

The forms of numeric constants are described in words in §4 *Data Types and Constants* of the f77 specification. "Appendix A1.1" provides a formal description. We define the set of constants of for each type, for example for integers as *IntegerConstants*, and thus the typing rule is smily that a constant belongs to a given set.

4.4.2 Scalars

The typing rule for a scalar s is simply that any access of a scalar variable, this variable must have the same type, which must be the type from its declaration in the current context (code unit) Γ and be a scalar. In a Fortran expression, this means that all variables with a type statement in a code unit will be of the type determined by that statement.

4.4.3 Arrays

The typing rule for an array declaration is that in addition to being of a valid Scalar type

- It must have a valid dimension attribute $d = ((b_1, e_1), \ldots, (b_i, e_i), \ldots, (b_k, e_k))$
- And for any array access $a(j_1, \ldots, j_k), j_i \in [b_i, e_i]$
  - The number indices must $k = \#d$,
  - The type must be the scalar type

Note that the additional condition of validity of the range of the array indices $a(j_1, \ldots, j_i, \ldots, j_k)$, $\forall\, i \in [1, k] | j_i \in [b_i, e_i]$ is not a type checking condition but an run-time range checking condition, so it is not part of the typing rules.

Array slicing Fortran 90 allows array slicing using the notation $(b_s : e_s : s_s)$, and it is quite common in FORTRAN 77 style code. For example:

Example 1 Array slicing

```
integer a, s
dimension a(5,7), s(3)
s = a(2,1:5:2)
```

The array s will be populated with the values from *a(2,1)*, *a(2,3)* and *a(2,5)*. From a type checking perspective, we need to check if the slice has the correct type, in this case the same type as the array to which it is assigned.

For a given tuple (b_i, e_i) from d, a slice is valid (i.e. within bounds) if $b_s \geq b_i, e_s \leq e_i, s_s \leq e_i - b_i$. We will call the set of indices in the slice a *DimSlice*, and it is given by

$$DimSlice = \{idx | idx \in [b, e] \wedge (idx - b)\, mod\, \}$$

, and we'll denote this as *DimSlice b e s*.

The type of a sliced array is determined from the *DimSlice* as follows:

- let the array a has *Dim d*, $d = (p_1, \ldots, p_i, \ldots, p_k)$, and we slice the tuple p_i with a valid slice $s_i = DimSlice\, b_s e_s s_s$.
- Then this results in a new $p_i' = (1, \#s_i)$
- and therefore a new *Dim d'*, $d' = (p_1, \ldots, p_i', \ldots, p_k)$

- and thus a new array type with the same scalar type as a and dimension d'

To be type safe, the size of the *DimSlice* must be known at type check time. This implies that the components *b, e, s* of the slice must be constant. If so, we can determine the size of the slice and thus check that the new type is correct given the context. In practice, our compiler performs aggressive linear constant folding, which means that an linear expression with constants as leaf nodes will be reduced to a constant. If the size of the *DimSlice* is only known at run time, our compiler allows to insert run-time checks as explained in "Appendix A2.1".

Arrays as indices Fortran also allows arrays to be indexed by other arrays, for example

Example 2 Arrays as indices

```
integer a(5,5), b(3), k(3)
k = (/ 1, 5, 2 /)
b = a(2, k)
```

The array *b* will contain the values from *a(2,1)*, *a(2,5)* and *a(2,2)* in that order.

The array to be used for indexing must be an array of rank 1 that contains the indices of the locations to be accessed, just like a *DimSlice*. Thus, the requirement for type safety is the same (the size of the array), and the criterion for the array to be valid for indexing is that all elements must be in the valid index range for the given array index. If the index range is only known at run time, our compiler allows to insert run-time checks as explained in "Appendix A2.2".

Bounds checking Fortran checks constant array bounds at type check time, and our compiler performs a more aggressive constant folding so that any index reducible to a constant with linear arithmetic will be considered constant. However, in general array indices are not known at type check time and therefore even in a well typed program, it is still possible to have out-of-bound errors. Fundamentally, index checking is not type checking because it concerns the actual values and has to be performed at run time. All modern Fortran compilers provide this option, e.g. `-fcheck=bounds` in the GNU gfortran.

4.4.4 Subroutines and functions

As explained in Sect. 4.2, every Fortran subroutine or external function can be transformed into a pure function. Intrinsic functions are pure by definition. From a type checking perspective, the difference between a Fortran function (external or intrinsic) and subroutine is that a function call can occur in an expression, and therefore has a return type, and a subroutine call is a statement and so has no return type. As explained before, we consider both subroutines and functions to be pure in the sense that any interaction with the code is via the arguments and return value.

- The subroutine declaration typing rule is that every dummy argument must be of a valid Fortran type. A subroutine does not return a type, we denote this by using *void*.

- The subroutine application (`call`) typing rule is that every call argument and every dummy argument must have the same type. Because a subroutine call is a statement, it does not return a type.
- The external function declaration typing rule is that every dummy argument must be of a valid Fortran type. An external function must have a return a type.
- The function application typing rule is also that every call argument and every dummy argument must have the same type. In that case, the function application is of the type of the return type.

Higher-order functions In the above, we have glossed over one important detail: subroutines and external functions can take the names of other subroutines or external functions as arguments. The case of an external functions is covered by the above typing rules because the function passed as argument has as type the return type and as such is indistinguishable from a variable. A subroutine passed as an argument however does not have a type, so we have to add *void* to the set of types that can are valid for arguments of a subroutine or function.

The f77 specification §8.7 *EXTERNAL Statement* requires that any external function or subroutine used as an argument is declared using the EXTERNAL statement. Omitting this declaration results in a type error.

Strictly speaking this means that *External* is a type attribute for functions or subroutines. However, for the purpose of this paper we will not extended type but instead group the EXTERNAL functions in a separate context. The actual type checking of higher-order functions is not possible at compile time. In "Appendix A2.3", we present an algorithm for run-time type checking via the construction of a sum type of the higher-order functions in a compute unit.

4.4.5 Assignments

Because in Fortran the assignment is a statement, it does not return a type. Therefore, the type check rule for an assignment of a variable v declared in the code unit Γ to an expression e which may contain any variable v_i declared in the code unit Γ is that the variable and the expression must be of the same type.

According to the f77 specification, only assignments to variables and array elements are valid, but the extension to arrays as in Fortran 90 is very common. The above typing rule does not limit the type check to scalars, so array assignments will type check if the types match.

4.4.6 Expressions

Expressions can consist of constants, variables, operators and calls to intrinsic or external functions.

Polymorphic numeric operators Numeric operators in Fortran are *polymorphic*, i.e. they can handle operands of any numeric type. We write 'any numeric type' using a type variable and a constraint: *Num a*

- The operators $+,-,*$ have type:

$$Num\, a \Rightarrow a \to a \to a$$

For any type a that is a valid numeric type (see Definition 1 in "Appendix A1.1"), the operator takes two arguments of that type a and returns an argument of that type a.

- The operator `**` also has the type

  $$Num\, a \Rightarrow a \to a \to a$$

  except when the exponent is an integer, in which case the type is:

  $$Kind\, k,\ Num\, a \Rightarrow a \to Integer * k \to a$$

  i.e. raising any numeric type to an integer power preserves the type.
- The unary − operator is also polymorphic with type

  $$Num\, a \Rightarrow a \to a$$

  - In Fortran 90, all the above operators also work on arrays
  - Comparison operations .lt., .le.,.eq.,.ne.,.gt.,.ge. are all of type

    $$Num\, a \Rightarrow a \to a \to Bool$$

Polymorphic intrinsics Many intrinsic functions are also polymorphic. For the types of intrinsic functions, we refer to Table 5 in the f77 specification.

- Intrinsics are either of type

  $$Num\, a \Rightarrow a \to a$$

  except for

  $$\texttt{imag} : Kind\, k \Rightarrow Complex * k \to Real * k$$

  or of type

  $$Num\, a \Rightarrow a \to a \to a$$

  except for

  $$\texttt{dprod} : Real * 4 \to Real * 4 \to Real * 8$$

- The intrinsics min and max take a list of arguments of a given type and undetermined length (denoted by […])

  $$Num\, a \Rightarrow [a] \to a$$

  except

  $$\texttt{amin0}, \texttt{amax0} : [Integer * 4] \to Real * 4$$
  $$\texttt{min1}, \texttt{max1} : [Real * 4] \to Integer * 4$$

Expression type rule

The expression forms a tree of applications of either operators or intrinsic functions, with the leaves being constants or variables. Type checking is performed via recursive descent.

4.4.7 Type conversions for polymorphic operators

The f77 specification defines specific intrinsic functions `int`, `real`, `dble` and `cmplx` for the purpose of type conversion (Table 5), and the Fortran 90 specification extends these to include the *kind* (thereby making `dble` redundant). Their signatures are, respectively:[15]

$$\begin{aligned}\texttt{int} &: Num\, a, Kind\, k \Rightarrow a \rightarrow k \rightarrow Integer * k\\ \texttt{real} &: Num\, a, Kind\, k \Rightarrow a \rightarrow k \rightarrow Real * k\\ \texttt{cmplx} &: Num\, a, Kind\, k \Rightarrow a \rightarrow k \rightarrow Complex * k\end{aligned}$$

As the name and the kind argument identify a Num type, for the typing rules we use the generic notation[16]

$$cast\langle\tau_2\rangle : Num\, \tau_1, \tau_2 \Rightarrow \tau_1 \rightarrow \tau_2$$

Fortran allows implicit type conversions (coercion) for operators and assignments, according to some simple *subtyping* rules.

A type $\tau_1$is a subtype of a type τ_2 if it is safe to use a term of type τ_1 in an context that expects a term of type τ_2. We denote this as $\tau_1 <: \tau_2$.

The following (transitive) subtyping relations apply to numeric Fortran types:

$$\begin{aligned}Integer * k <: Real * k, k \in Kind\\ Real * k <: Complex * k\\ t * k_1 <: t * k_2, k_1 < k_2 \in Kind, t \in NumType\end{aligned}$$

Therefore, we can generalise the type conversion rules from Table 2 *Type and Interpretation of Result for* $x_1 + x_2$ of the f77 specification formally as:

if$\tau_1 <: \tau_2$then the type conversion rule is $cast\langle\tau_2\rangle\, e_1\ op\ e_2$ so the type of the expression is τ_2; if $\tau_2 <: \tau_1$ then it is $e\ op\ cast\langle\tau_1\rangle\ e_2$ and the type of the expression is τ_1, with $op = +,-,*,/$ or $**$.[17]

As for the relational operators .lt., .le., .eq., .ne., .gt., .ge., according to the f77 specification §6.3.3 *Interpretation of Arithmetic Relational Expressions*:

[15] The actual signature for `cmplx` is more complicated because it can be used to construct a complex number from two reals, but for the purpose of type conversion, the presented signature suffices.

[16] In Fortran 90, the type conversion functions can take an array operand: $cast\langle\tau_2\rangle : Num\, \tau_1, \tau_2, Dim\, d \Rightarrow Array\, \tau_1\, d \rightarrow Array\, \tau_2\, d$.

[17] Unless the exponent of `**` is an integer, in which case there is no type conversion.

> If the two arithmetic expressions are of different types, the value of the relational expression `e1 relop e2` is the value of the expression `((e1) - (e2)) relop 0`

Therefore, the type conversion rules are very similar, the only difference is that the type of the expression is Bool.

4.4.8 Type conversion of assignments

For assignments, the f77 specification states that the assignment is typed according the following rules:

> Execution of an arithmetic assignment statement causes the evaluation of the expression e by the rules in Sect. 6, conversion of e to the type of v , and definition and assignment of v with the resulting value, as established by the rules in Table 4.

This means that *e* is implicitly converted to the type of *v* even if the conversion is unsafe. Strictly speaking, this is a type error, and if we type check e.g. Example 3 using the GNU Fortran compiler with flags as shown in Example 4, we do indeed get a type error.

Example 3 Unsafe coercion

```
program unsafeCoercion
 integer i1,i2
 real r1
 r1 = 0.14159
 i1=3
 i2 = i1+r1
end
```

Example 4 Output from g77 for program `unsafeCoercion`

```
g77 -fsyntax-only -Werror=conversion test_type_coercion.f
test_type_coercion.f:6:11:
6 | i2 = i1+r1
  | 1
Error: Possible change of value in conversion from REAL(4) to INTEGER(4) at (1) [-Werror=conversion]
```

However, this behaviour is so common that by default, Fortran compilers only warn about unsafe conversions, and then only when warnings are enabled. Our compiler warns by default and converts the implicit type conversion to an explicit conversion as shown in Example 5. Thus the resulting code is type safe, and we assume that the explicit conversion is what the programmer wants.

Example 5 Explicit conversion

```
program unsafeCoercion
 integer :: i1,i2
 real :: r1
 r1 = 0.14159
 i1=3
 i2 = int(i1+r1,4)
 end program unsafeCoercion
```

4.5 Conclusions regarding the type safety of the Fortran type system

Based on the above analysis, we conclude that FORTRAN 77 programs that are explicitly typed and consist of pure functions are type safe, except for three specific constructs: array slicing and array indexing with values that are unknown at compile time and higher-order functions. Current compilers guarantee type safety if the slice indices or the arrays used as index are constant, and also if the array used for indexing is of the wrong rank. However, if the indices are non-constant expressions, potentially unsafe programs pass without warning or error. Our compiler will issue a type error, which can be relaxed to warning or run-time type check, rather than ignoring the potential unsafe behaviour.

Calls to functions passed as arguments to other functions (i.e. higher-order functions) are fundamentally unsafe because the type signature of Fortran functions and subroutines does not contain the information about the types of the arguments. We present a novel run-time check which is equivalent to constructing a sum type for all external functions and type checking the variants.

In principle, some type coercions are also unsafe. However, unsafe type coercions are recognised by current compilers, and the compiler can produce a warning or error if the option is enabled. So type coercions don't compromise the type safety. Our compiler follows this convention.

In conclusion, FORTRAN 77 programs that are explicitly typed and consist of pure functions are almost entirely type safe at compile time and can be made entirely type safe through the addition of run-time type checks for array slicing and array indexing with values that are unknown at compile time and higher-order functions.

In the next sections, we discuss the transformations required to ensure that the resulting programs are explicitly typed and consist of pure functions.

5 The problem for type safety: loss of type information

From the perspective of this paper, the main problem with COMMON blocks and EQUIVALENCE associations is that they are not type safe: the f77 specification does not mention any typing rules and in practice, any datum stored in a common block loses all type information.

This means in particular that there is no type coercion between real (and by extension complex) and integer values in COMMON blocks. The same is true for EQUIVALENCE statements: they associate different names with the same memory location, but the type of the word written to the memory location is erased. Therefore, the following is legal and does not generate any warnings, but is incorrect

Example 6 Loss of type information in EQUIVALENCE

```
integer*4 i1
real*4 r1
equivalence (i1,r1)
i1 = 42
print *, r1 ! prints 5.88545355E-44
r1 = 42
print *, i1 ! prints 1109917696
```

What happens is that there is a sequence of four bytes stored in memory and referenced by both *i1*, which results in it being interpreted as a 32-bit signed integer in 2's complement format, and *r2*, which results in it being interpreted as a 32-bit real, i.e. an IEEE 754 single-precision floating point number. There is no information in the sequence of four bytes to indicate which interpretation is the correct one.

6 Program transformations for type safety

In the preceding sections, we analysed the type safety of a FORTRAN 77 program that consists of pure functions and where all variables, parameters and functions are explicitly typed. In this section, we show how any FORTRAN 77 program can be transformed into an equivalent program with these properties. First, we show how to transform side-effect-free procedures into pure functions. Then we discuss how to remove COMMON blocks in a type-safe manner. Because of the assumptions that our procedures do not contain I/O calls or external library calls, the COMMON blocks are the only source of potential side effects.

6.1 Transforming side-effect-free Fortran subroutines into pure functions

A side-effect-free FORTRAN 77 subroutine can be translated into a pure function as shown in Algorithm 1, which is linear in the number of subroutine arguments:

Algorithm 1: Transforming a FORTRAN 77 procedure into a pure function

- Infer the INTENT of every argument (see Section 6.2)
- Replace every InOut argument with an (In,Out) tuple
- Every read is from the In argument, every write is to the Out argument

Thus,
`subroutine` $f(a_1, ..., a_i, ..., a_n)$
τ_i,`intent(InOut)`:: a_i
$(..., a_i, ..., ..., l_i, ...) = exp(..., a_i, ..., ..., l_i, ...)$
`end subroutine` f

becomes

`subroutine` $f'(a_1, ..., a_{i,in}, a_{i,out}, ..., a_n)$
τ_i,`intent(In)`:: $a_{i,in}$
τ_i,`intent(Out)`:: $a_{i,out}$
τ_i :: a_i`! local`
$a_i = a_{i,in}$
$(..., a_i, ..., ..., l_i, ...) = exp(..., a_i, ..., ..., l_i, ...))$
$a_{i,out} = a_i$
`end subroutine` f'

Denoting the transformed argument list as *a'*:

$$(\forall a_i \in a' \,|\, \text{intent}(a_i) = \text{InOut}) = \emptyset$$
$$y = (\forall a_i \in a' \,|\, \text{intent}(a_i) = \text{Out})$$
$$a'' = a' \setminus y \iff \forall a_i \in a'' : \text{intent}(a_i) = \text{In}$$

In words, *a'* does no longer contain any element with INTENT InOut; *y* is the tuple of all elements from *a'* with intent Out and *a"* is the tuple of all arguments of *a'* with INTENT In. In this way, we have identified the function arguments and the function return value and their types. So regardless of the subroutine syntax, with this information it is now a pure function as far as its arguments are concerned. For an external function, the algorithm is the same but the return value tuple includes the original return value.

6.2 Inferring the INTENT of procedure arguments

Because the subroutines to be offloaded cannot contain external calls or I/O calls, once all COMMON variables have been transformed into subroutine arguments, we can infer the INTENT of all procedure arguments by recursive descent into nested calls, as shown in Algorithm 2:

Algorithm 2: Infering INTENT

1. Determine the INTENT for all leaf subroutines:
   - Using a recursive descent of the call graph from the entry procedure p_e until a leaf procedure p_l (one that does not call other procedures) is reached.
   - All different paths through the call graph need to be followed in the order they are called.
   - When a leaf node is reached, determine the INTENT of the arguments of the leaf subroutine using Algorithm 3.
2. Using recursive descent, determine the INTENT of all arguments of the calling subroutines.

The INTENT reflects if a subroutine argument is accessed read-only (*In*), write-only (*Out*) or read-write (*InOut*) in the subroutine. To determine the INTENT of an argument in a leaf subroutine, we use Algorithm 3:

- Inspect every statement that accesses one or more of the subroutine arguments (i.e. all expressions and procedure calls, including intrinsic calls) in order of occurrence.
- Based on the type of statement, it is possible to determine how a variable is accessed (*Read*, *Write* or *Read-Write*).
  - Initially, the INTENT of an argument is *Unknown* because the f77 specification does not support the INTENT attribute.
  - Based on the access pattern in the subroutine, set the INTENT to In, Out or InOut.
  - Once an INTENT has been set to InOut, there is no need to look at any remaining statements.
  - If the INTENT is set to In or Out, further statements can result in a change to InOut. In that case,inspect all further statements in the subroutine.
- The INTENT of an argument is determined based on its access in a statement using Algorithm 4.

Algorithm 3: Infering Intent for leaf subroutine

Input: INTENT = Unknown

while *INTENT != InOut* **do**
  Determine INTENT using Algorithm 4.
end

Output: INTENT for leaf subroutine argument

The combined algorithm has linear complexity in terms of the total number of nodes in the call tree and the number of arguments of each subroutine.

6.3 Transforming IMPLICIT typing into explicit typing

According to §4.1.2 *Type Rules for Data and Procedure Identifiers* of the f77 specification, in FORTRAN 77 a variable

> may have its type specified in a type-statement (8.4) as integer, real, double precision, complex, logical or character. In the absence of an explicit declaration in a type-statement, the type is implied by the first letter of the name. A first letter of I, J, K, L, M or N implies type integer, and any other letter implies type real, unless an IMPLICIT statement (8.5) is used to change the default implied type.

An IMPLICIT statement specifies a type for all variables that begin with any letter that appears in the specification. From a type safety perspective, the problem with this typing discipline is no *referential transparency*, i.e. if the name of a variable changes then the result of a computation may change. As our aim is to create pure functional code, our compiler infers explicit type declarations ("type-statements" in the f77 specification) for all implicit typed variables.

The algorithm for this (Algorithm 4) is straightforward. It is linear in terms of the number of undeclared variables in the code unit.

Algorithm 4: Infering Intent for leaf subroutine

```
Input: current INTENT
if access = Read-Write then
  | INTENT = InOut
else if access = Read then
  |  if INTENT = Unknown then
  |    | INTENT = In
  |  else if INTENT Out then
  |    | INTENT = InOut
else if access = Write then
  |  if INTENT = Unknown then
  |    | INTENT = Out
  |  else if INTENT In then
  |    | INTENT = InOut
Output: updated INTENT
```

6.4 Transforming COMMON blocks into procedure arguments

The f77 specification defines the semantics of the COMMON statement in §8.3 COMMON *Statement*:

> The COMMON statement provides a means of associating entities in different program units. This allows different program units to define and reference the same data without using arguments, and to share storage units.

The f90 specification (§5.5.2 COMMON *statement*) has a slightly different wording:

> The COMMON statement specifies blocks of physical storage, called common blocks, that may be accessed by any of the scoping units in a program. Thus, the COMMON statement provides a global data facility based on storage association.
> Storage sequences are used to describe relationships that exist among variables, common blocks and result variables. Storage association is the association of two or more data objects that occurs when two or more storage sequences share or are aligned with one or more storage units.

As explained above, the main reason to remove COMMON blocks is to create pure functions that don't share a memory space with their caller code unit. This is an essential requirement for offloading to accelerators. However, type-safe removal of COMMON blocks and rewriting of EQUIVALENCE statements also guarantees that the resulting code is type-safe.

Our approach is to convert COMMON block variables into subroutine arguments. The more common approach of conversion into module-scoped variables is not suitable for our purpose because it does not result in pure functions. (Furthermore, because of the difference in semantics of storage association and module scoped variables, this approach only works for COMMON blocks where all variables are aligned, whereas COMMON blocks allow overlapping sequences). One of the main contributions of this paper is in this conversion and the associated type checks, presented in the next section (Sect. 7).

In the following sections, we use suffix or subscript c to indicate variables from the caller and l to indicate variables local to the callee.

6.4.1 Construct the COMMON block chain

In a subroutine call chain, it is not necessary for a COMMON blocks to occur in the caller. It is sufficient that the COMMON block used in a called subroutine occurs somewhere in the call chain. As a consequence, it is not generally possible to associate the COMMON block variables in a called subroutine with those of the caller. For example:

Example 7 COMMON block chain

```
program ex1
  common /bf2/x
  common /bf1/y
  call f1
end program ex1
subroutine f1
  common /bf1/y1
  call f2
end subroutine f1
subroutine f2
```

```
  common /bf2/x2
end subroutine f2
```

In this example, x2 in f2 is associated with x in the main program

Example 8 Passing arguments through the call chain

```
program ex1
  call f1(x,y)
end program ex1
subroutine
  f1(x,y1)
  call f2(x)
  ! ...use y1 ...
end subroutine f1
subroutine f2(x2)
  ! ...use of x2
end subroutine f2
```

So, the argument for f2 has to be passed via f1 from the main program. Therefore, we need to analyse the code for the call chain paths between disjoint COMMON blocks and pass all arguments via the intervening calls. This also requires checking if the names are unique an renaming if necessary. The result of this analysis is that for every called subroutine, we have a pair consisting of the common block sequence that will become the call arguments, and the common block sequence that will become the dummy arguments.

6.4.2 Associate COMMON block variables in procedure calls with the caller

To create the call arguments and dummy arguments, we need to identify which variable in the caller sequence matches which in the subroutine call sequence (called the 'local' sequence for brevity). This is complicated by the fact that storage sequences are allowed to overlap and do not follow the normal type checking rules. For example, the following is acceptable:

Example 9 Overlapping sequences

```
! caller
real xc(8),z1c,z2c
complex yc
common yc,xc,z1c,z2c
! local real xl(2),zl(4)
complex yl(3)
common yl,xl,zl
```

The COMMON statements in Example 9 leads to following associations:

`complex yl(1)`	`complex yc(1)`
`complex yl(2)`	`real xc(1), xc(2)`
`complex yl(3)`	`real xc(3), xc(4)`
`xl`	`real xc(5:6)`
`zl(1:2)`	`real xc(7:8)`
`zl(3)`	`real z1c`
`zl(4)`	`real z2c`

Given that the associations type check correctly, then it follows that for every variable in a COMMON block declared in the caller, there is a corresponding variable in the called subroutine. In practice, these variables can be either arrays or scalars. Whereas for the purpose of type checking, we have assumed that all variables are scalar, again without loss of generality, we will now assume that all variables are arrays. A scalar s is simply syntactic sugar for the first element of an array of size 1, $s(1)$. This is merely to keep the rules more compact. As before, we traverse every array using a linear index starting at 1.

Because it is possible for arrays from the caller and arrays in the called subroutine to overlap in both directions, our strategy for converting the COMMON variables into dummy parameters is as shown in Algorithm 5. This algorithm is linear in terms of the total number of matching sequences in all common blocks, which is of the same order as the number of variables in the blocks but much smaller than the total storage size of the blocks.

Algorithm 5: Removal of IMPLICIT typing rules

1. Parse all IMPLICIT statements and turn them into a lookup table *{Char ⇒ (Type,ArrayOrScalar,Attribute)}*. This lookup table is initially populated with the default rule:
   `implicit integer (i-n), real (a-h, o-z)`
2. Analyse all executable statements for occurrences of undeclared variables and add declarations following the IMPLICIT rules, i.e
   (a) Get the first character
   (b) Look up the type in the lookup table
   (c) Create a type declaration
3. Add the created declarations before the first executable statement
4. Add IMPLICIT NONE before the first non-executable statement to specify that no implicit typing should be done in the generated code

The compiler maintains a global state record. The state information of each subroutine, *st*, is used in `updateDim` for evaluation of the array bounds.

the sequence *cseq* consists of tuples of the type declaration *decl* and the linear index *idx* in the COMMON block

$$cseq = (decl_1, idx_1), \ldots, (decl_i, idx_i), \ldots)$$

For every array variable, the type declaration contains a *Dim* field *d* which is an array of *(start index, end index)* tuples, with total size *sz*.

The list of equivalence pairs *eqps* contains the matched up declarations of the original COMMON block variables of the called subroutine and the COMMON block variables of the caller that constitute the new arguments to the subroutine. The caller variables are prefixed with the name of the block and the caller subroutine.

The matching algorithm traverses the local sequence $cseq_c$ and matches each element to one or more elements of the caller sequence $cseq_l$. The algorithm (Algorithm 6) is iterative and stops when the local sequence has been consumed and returns *eqps*.

In the calls to `updateDim`, the subscript e in $idx_{l|c,e}$ indicates the end of the common block sequence; 1 is the start of the sequence. The • separates an element from the rest of a list. On the left-hand side, it means the element is removed from the list, on the right-hand side it means the element is added.

Algorithm 6: strategy for converting the COMMON variables into dummy parameters is as follows

- Declare dummy parameters with the names of the variables in the caller (prefixed with the name of the caller and the COMMON block).
- Determine the assignments required to match these dummy parameters with the variables that used to be COMMON block variables in the caller subroutine, but now are ordinary local variables.
- Insert these assignments after the last specification statement.
- At the end of the subroutine, insert the corresponding reverse assignments.
- If required, the right-hand side of the assignment will contain an explicit cast and/or a reshape instruction, because it is possible that the arrays in the caller and the called subroutine have different shapes.

6.5 Removal of EQUIVALENCE statements

According to §8.2 EQUIVALENCE *Statement* of the f77 specification,

> An EQUIVALENCE statement is used to specify the sharing of storage units by two or more entities in a program unit. This causes association of the entities that share the storage units. If the equivalenced entities are of different data types, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If a variable and an array are equivalenced, the variable does not have array properties, and the array does not have the properties of a variable.

This is another form of storage association, with the same issue that it is not type-safe. For example, the equivalence between *vb2* and *d1* or *r1* in Example 10 is a type error but passes silently because the f77 specification does not mandate an error or even a warning.

Example 10 Unsafe equivalence

```
program test_equivalence
 implicit integer (i,v)
 dimension vb2(8)
 logical l2(8)
 double precision d1(4)
 real*4 r1(8)
 ! not type safe, in fact plain wrong
 equivalence (vb2,d1)
 equivalence (vb2,r1)
 ! This is OK
 equivalence (vb2,l2)
 equivalence (r1,l2)
 ...
end
```

Therefore, EQUIVALENCE statements also need to be refactored. They come with their own flavour of complications.

6.5.1 Transitivity

If a line has multiple tuples, a variable can occur in more than one tuple, e.g.:

```
(v1,v2),(v2,v3)
```

So, we must effectively do a transitivity check across all tuples. We do this by checking if an element of a tuple occurs in another tuple. It is sufficient to do this for a single element because the transitivity means that every element from the one tuple will be associated with every element from the other.

6.5.2 Quadratic complexity

Furthermore, the tuples (called lists in the spec) can have more than two elements.

```
(v1,v2,v3),(v3,v4,v5)
```

then this effectively means `(v1,v2,v3,v4,v5)`, and each of these variables is associated with all the others, so there are 10 unique associations in this example. In general, for a tuple of n values, there will be $(n-1).n/2$ associations.

This algorithm therefore has quadratic complexity but fortunately the number of associated variables in a program is never very large.

6.5.3 Overlapping ranges

A final complication is that overlapping is allowed, e.g.:

Example 11 Overlap in EQUIVALENCE

```
dimension rade11(5), rade12(5)
equivalence (rade11(4), rade12(2))
```

Because the arrays start at 1, and they overlap, this actually creates an equivalence between `RADE11(3), RADE12(1)` and `RADE11(5), RADE12(3)` as well. So, we have to equate the overlapping ranges.

6.5.4 Equivalence pairs

Taking the above into account, we can create a set of pairs identifying the scalar variables or array accesses that are equivalent. We call these "equivalence pairs". To be able to remove the EQUIVALENCE statements, we must insert additional assignment after every statement where one of more of the variables that is part of an equivalence pair gets modified. The possible cases are

- If the one of the variables in an equivalence pair, v_l , is local, and the other is an argument a that results from refactoring a COMMON block, then we need an initial assignment $v_l = a$.
- Any variable on the left-hand side of an assignment
- Any argument to a subroutine or function call that is used with INTENT Out or InOut.

The construction if the equivalence pairs is shown in Algorithm 7. As before, we consider array accesses as syntactic sugar for indexed scalars, so that we don't need to distinguish between scalars and arrays. The algorithm constructs a set of pairs *EquivalencePairs* from the original tuples in the EQUIVALENCE statements, taking into account transitivity. It then groups the pairs into pairs tuples *EquivalenceSets*. The set *EquivalenceVars* is the set of the first element of each pair.

Algorithm 7: Matching up COMMON block variables in a subroutine call

Input: $st_l, st_c, cseq_l, cseq_c$

while $cseq_l \neq \emptyset$ **do**

$$(decl_l, idx_l) \bullet cseq_l = cseq_l \qquad (1)$$
$$(decl_c, idx_c) \bullet cseq_c = cseq_c$$

$$d_l = \begin{cases} sz_l - idx_l > sz_c - idx_c & \text{updateDim } st_l\, d_l\, idx_l\, 1 \\ sz_l - idx_l <= sz_c - idx_c & \text{updateDim } st_l\, d_l\, idx_l\, idx_{l,e} \end{cases} \qquad (2)$$

$$d_c = \begin{cases} sz_l - idx_l => sz_c - idx_c & \text{updateDim } st_c\, d_c\, idx_c\, 1 \\ sz_l - idx_l < sz_c - idx_c & \text{updateDim } st_c\, d_c\, idx_c\, idx_{c,e} \end{cases} \qquad (3)$$

$$cseq_l = \begin{cases} sz_l - idx_l > sz_c - idx_c & \begin{cases} sz_l - idx_{l,e} \geq 1 & (decl_l, idx_l + 1) \bullet cseq_l \\ sz_l - idx_{l,e} < 1 & cseq_l \end{cases} \\ sz_l - idx_l \leq sz_c - idx_c & cseq_l \end{cases} \qquad (4)$$

$$cseq_c = \begin{cases} sz_l - idx_l < sz_c - idx_c & \begin{cases} sz_c - idx_{c,e} \geq 1 & (decl_c, idx_c + 1) \bullet cseq_c \\ sz_c - idx_{c,e} < 1 & cseq_c \end{cases} \\ sz_l - idx_l \geq sz_c - idx_c & cseq_c \end{cases} \qquad (5)$$

$$eqps = eqps \bullet (decl_l \{Dim = d_l\}, decl_c \{Dim = d_c, Prefix = p\}) \qquad (6)$$

end

Output: $eqps$

The algorithm to replace the EQUIVALENCE statements by assignments is shown in Algorithm 8. It is linear in the number of occurrences in the code unit of variables that occur in the set of equivalence pairs.

Algorithm 8: Creating EQUIVALENCE pairs

```
Input: EquivalenceSets = ()
Input: EquivalenceVars = ()
Input: EquivalencePairs = ()
for t ∈ EquivalenceTuples do
    for v1 ∈ t do
        for v2 ∈ t do
            if v1 ≠ e2 then
                EquivalencePairs = EquivalencePairs • (v1, v2)
            end
        end
    end
    transitive = False
    for v ∈ t do
        if v ∈ EquivalenceVars then
            trans = True
            v_trans = v
            Stop
        end
    end
    if transitive then
        for v1 ∈ t do
            if v1 ≠ v_trans then
                for v2 ∈ EquivalenceSets(v_trans) do
                    EquivalencePairs = EquivalencePairs • (v1, v2)
                end
            end
        end
    end
    for (v1, v2) ∈ EquivalencePairs do
        EquivalenceSets(v1) = (EquivalenceSets(v1) • v2)
        EquivalenceSets(v2) = (EquivalenceSets(v2) • v1)
        if v1 ∉ EquivalenceVars then
            EquivalenceVars = (EquivalenceVars • v1)
        end
        if v2 ∉ EquivalenceVars then
            EquivalenceVars = (EquivalenceVars • v2)
        end
    end
end
Output: EquivalenceSets, EquivalenceVars
```

6.6 Summary of program transformations

The algorithms presented in this section transform the code units of a FORTRAN 77 program into side-effect-free pure functions. All arguments and local variables are explicitly, statically typed using the algorithm from Sect. 6.3.

As a prerequisite to the actual transformation into pure functions (Sect. 6.1), we inferring the INTENT of procedure arguments using the algorithm from Sect. 6.2.

To ensure type safety of the transformations, the type checks presented in Sect. 7 are performed before the COMMON and EQUIVALENCE statements are eliminated using the algorithms presented in Sects. 6.4 and 6.5.

7 A novel type checking algorithm for COMMON blocks and EQUIVALENCE associations

In Sect. 4, we analysed the type safety of a FORTRAN 77 program that consists of pure functions and where all variables, parameters and functions are explicitly typed. In Sect. 6, we have presented the algorithms to transform any FORTRAN 77 program into an equivalent program with these properties. In particular, we discussed how to remove COMMON blocks and EQUIVALENCE associations. In this section, we present a novel algorithm to type check variables in COMMON and EQUIVALENCE statements. If the type check passes, then COMMON and EQUIVALENCE statements can be safely removed. Otherwise, it means the code is not type safe and in practice most likely incorrect. If the original code passes the type check then the transformed code without COMMON and EQUIVALENCE statements will be type safe when checked with any of the major Fortran compilers.

As discussed in Sect. 5, COMMON blocks and EQUIVALENCE associations simply associate memory storage with variable names, but do no preserve the original type information, nor do they attempt type conversion. Because of this, the type checks presented here are different from the type checks discussed in Sect. 4.

For the purpose of type checking, and without loss of generality, we assume that all variables are scalar: an array a is considered as syntactic sugar for an ordered collection of scalars with names $a(i)$. We further assume all arrays are linear and traversed using an index starting at 1.

The rules for type soundness of COMMON and EQUIVALENCE statements are shown in Algorithm 9. The suffix l refers to the sets and variable local to a procedure, the suffix c to variable in the code unit of the the caller of the procedure. This algorithm has linear complexity in terms of the total number of variables in all common blocks in a program.

Algorithm 9: Replacing EQUIVALENCE statements with assignments

```
Input: EquivalenceSets =
       (v_{1,l} ⇒ (v_{1,r_1}, v_{1,r_2}, ...), ..., v_{i,l} ⇒ (v_{i,r_1}, ..., v_{i,r_j}, ...), ...)
Input: EquivalenceVars = (v_{1,l}, ..., v_{i,l}, ...)
Input: Statements(c)
for stmt ∈ Statements(c) do
    if stmt is Equivalence(v, a) and a ∈ Args(h) and v ∉ Args(h) then
        Statements(c) = Statements(c) • v = a
    end
    if stmt = Assignment(v, RHSexpr) then
        if v ∈ EquivalenceVars then
            for v_r ∈ EquivalenceSets(v) do
                Statements(c) = Statements(c) • v_r = v
            end
        end
        if FunctionCalls(RHSexpr) ≠ ∅ then
            for f ∈ FunctionCalls(RHSexpr) do
                for a ∈ Args(f)|Intent(a) ≠ In do
                    if a ∈ EquivalenceVars then
                        for a_r ∈ EquivalenceSets(a) do
                            Statements(c) = Statements(c) • a_r = a
                        end
                    end
                end
            end
        end
    end
    else if stmt = SubroutineCall(s) then
        for a ∈ Args(s)|Intent(a) ≠ In do
            if a ∈ EquivalenceVars then
                for a_r ∈ EquivalenceSets(a) do
                    Statements(c) = Statements(c) • a_r = a
                end
            end
        end
    end
end
Output: Statements(c)
```

- Kind matching (*rule-kind*) A scalar type is atomic, and therefore we cannot split the type, which would be the case if we attempted to map types with different kinds. If K_c and K_l are the ordered sets of the kinds of all variables associated via the COMMON block then

$$K_{c,i} = K_{l,i}, \quad \forall i \in \#K_l$$

- No COMMON block extension in the called subroutine (*rule-size*) A word is a sequence of bytes. The ordered set of kinds indicates the size of each word in the ordered set of words in a COMMON block. The ordered set of words in a COMMON block accessed from a subroutine must be no larger than the size of

the COMMON block in the caller, because otherwise the caller would not have declared the corresponding typed variables. If W_l and W_c are the ordered sets of all words associated via the COMMON block then

$$\#W_l \leq \#W_c$$

- Logical coercion (*rule-logical*) The default rule (*rule-default*, see below) is that all types must match between the sequence of variables in the caller and the called subroutine. However, this rule is too strict: there are two cases in which type coercion is sound. Let $T_{c,i}$ and $T_{l,i}$ be the types of corresponding words in the ordered sets associated via the COMMON block. The first case involves logicals:

$$\begin{cases} T_{c,i} = \text{logical} \wedge T_{l,i} = \text{integer} \\ T_{c,i} = \text{integer} \wedge T_{l,i} = \text{logical} \\ T_{c,i} = \text{real} \wedge T_{l,i} = \text{logical} \end{cases}$$

  A logical is false when coerced from 0 and true otherwise. Therefore interpreting a logical as an integer gives 1 or 0, and interpreting an integer or real as a logical will return correct values of `.true.` or `.false.`. Therefore, interpreting a logical as a real is only correct for `.false.` because the value of `.true.` interpreted as a real is a non-zero number that depends on the kind of the real. As this is quite non-intuitive, our type checker therefore throws an error on attempts to interpret a logical as a real.
- Complex coercion (*rule-complex*) The second case involves complex numbers, which can be coerced to and from two contiguous real numbers:

$$\begin{cases} T_{c,i} = \text{complex} \wedge T_{l,i} = \text{real} \wedge T_{l,i+1} = \text{real} \\ T_{c,i} = \text{real} \wedge T_{c,i+1} = \text{real} \wedge T_{l,i} = \text{complex} \end{cases}$$

- Default rule (*rule-default*) In all other cases, the types must match:

$$T_{l,i} = T_{c,i} \quad , \forall i \in \#K_l$$

- Thus the overall type check rule becomes:

$$\text{rule-kind} \wedge \text{rule-size} \wedge (\text{rule-logical} \vee \text{rule-complex} \vee \text{rule-default})$$

With these type checking rules, we can type check the soundness of associations in COMMON blocks and EQUIVALENCE statements. If the associations are correctly typed, we can proceed to remove them as discussed in Sect. 6.1.

8 Conclusions

In this paper, we have formally analysed the type safety of FORTRAN 77 programs. We have shown that FORTRAN 77 programs that are explicitly typed and consist of pure, side-effect-free functions are type safe at compile time with the exception of array slicing and array indexing with values that are unknown

at compile time, and higher-order functions, and that even these features can be made entirely type safe through the addition of run-time type checks for these features.

We have presented the algorithms for transforming arbitrary FORTRAN 77 programs into explicitly typed, type-safe code consisting of pure, side-effect-free functions.

We have created a source-to-source compiler which implements the transformations and type checks presented in this paper and generates fully type safe Fortran 90 code.

That FORTRAN 77 programs can be made entirely type safe through program transformations is a significant finding in its own right. However, our work has considerable benefits. The obvious benefit of type safety is fewer errors. Furthermore, our compiler transform legacy FORTRAN 77 code into modern, type safe Fortran 90. And finally, because the resulting code consists of self-contained pure functions, each of these functions can also be offloaded more easily to accelerators such as GPUs or FPGAs. This is demonstrated by our work on automated parallelisation and GPU-offloading [20].

Appendix 1: Formal definition of the FORTRAN 77 type system and typing rules

A1.1: Set definitions of Fortran types and constants

With the conventions from Sect. 4.3, we can construct the sets of valid types for FORTRAN 77 using set theory.

Definition 1 The FORTRAN 77 type system

Type $= \{Integer, Real, Complex, Logical\}$

NumType $= \{Integer, Real, Complex\}$

Kind $= \{2^n \mid n \in [0, 5]\}$

Scalar $= \{Type \times Kind\}$, an element is denoted as *Scalar t k* or *t*k*

Num $= \{NumType \times Kind\}$, an element is denoted as *Num a*

Bool $= \{Logical \times Kind\}$, an element is denoted as *Bool*

Dim $= \left\{((b_i, e_i), \ldots, (b_i, e_i), \ldots, (b_k, e_k)), \forall k, i \in [1, 7], b_i, e_i \in \mathbb{Z}, b_i \leq e_i\right\}$, so Dim is a set of ordered sets of tuples, we denote an element as *Dim d*

Array $= \{Scalar \times Dim\}$, and we denote an element of this set as *Array (Scalar t k) (Dim d)*

FortranType $= Scalar \cap Array$

Tuple $= (\tau_1 \times \cdots \times \tau_i \times \cdots \times \tau_k), \forall\, i \in [1, k] \mid \tau_i \in FortranType$ and we'll write *Tuple t*, where $t = (\tau_1, \ldots, \tau_k)$

The forms of numeric constants are described in words in §4 *Data Types and Constants* of the f77 specification. Using Extended Backus-Naur Form (EBNF,[18]), we can describe them formally as show in Definition 2.

Definition 2 EBNF for numeric constants

integer-constant ::= [sign] {digit}+
sign ::= + | -
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
real-constant ::= [sign] {digit}* decimal-point {digit}* [real-exponent]
| [sign] {digit}+ [decimal-point {digit}*] real-exponent
decimal-point ::= .
real-exponent ::= E [sign] {digit}+
double-constant ::= [sign] {digit}* decimal-point {digit}* [double-exponent]
| [sign] {digit}+ [decimal-point {digit}*] double-exponent
double-exponent ::= D [sign] {digit}+
complex-constant ::== (real-constant , real-constant)
logical-constant ::== .TRUE. | .FALSE.
numeric-constant ::== integer-constant | real-constant | double-constant | complex-constant | logical-constant

We define the set of numeric constants in terms of the above:

$$NumConstants = \{n \mid n \text{ is a numeric constant}\}$$

The general form of a type τ in Fortran 77

Definition 3 General form a type τ in Fortran 77

$$\begin{array}{ll} \tau ::= & \\ \quad FortranType & \text{primitive type} \\ \quad \mid Tuple & \text{tuple type} \\ \quad \mid \tau \rightarrow \tau & \text{function type} \\ \quad \mid void & \text{non-type} \\ \quad \mid a & \text{type variable} \end{array} \tag{7}$$

A1.2: Fortran typing rules

With the above definitions for the types, the typing rules for FORTRAN 77, described and discussed in Sect. 4, can be formally expressed as follows:

[18] https://www.w3.org/TR/2008/REC-xml-20081126/#sec-notation.

$$\frac{}{n \ : \ Num}. \forall\, n \in NumConstants \quad [CONST]$$

$$\frac{}{\Gamma \vdash s \ : \ \tau_s = Scalar\, t\, k} \ [SCALAR]$$

$$\frac{}{\Gamma \vdash a \ : \ \tau_a = Array\,(Scalar\, t_a\, k_a)\ (Dim\, d)} \ [ARRAY\ DECL]$$

$$k = \#d$$

$$\frac{\Gamma \vdash a \ : \ \tau_a \quad \Gamma \vdash j_i \ : \ Integer, \forall\, i \in [1,k]}{\Gamma \vdash a(j_1, \ldots, j_i, \ldots, j_k) \ : \ \tau_s = Scalar\, t_a\, k_a} \ [ARRAY\ ACCESS]$$

$$\frac{}{FortranType \cap \{void\}\ \mathbf{T_k} \Rightarrow sf \ : \ \mathbf{T_k} \rightarrow void} \ [SUB\ DECL]$$

$$\frac{FortranType \cap \{void\}\ \mathbf{T_k} \Rightarrow sf \ : \ \mathbf{T_k} \rightarrow void \quad \Gamma \vdash \mathbf{e_k} \ : \ \mathbf{T_k}}{\Gamma \vdash \texttt{call}\ sf\ e_k \ : \ void} \ [SUB\ CALL]$$

$$\frac{}{FortranType\ \mathbf{T_k}, \tau_f \Rightarrow f \ : \ \mathbf{T_k} \rightarrow \tau_f} \ [FUN\ DECL]$$

$$\frac{FortranType\ \mathbf{T_k}, \mathbf{\tau_f} \Rightarrow f \ : \ \mathbf{T_k} \rightarrow \tau_f \quad \Gamma \vdash \mathbf{e_k} \ : \ \mathbf{T_k}}{\Gamma \vdash f\ \mathbf{e_k} \ : \ \tau_f} \ [FUN\ CALL]$$

$$\frac{\Gamma \vdash v \ : \ \tau \quad \Gamma(v_i) \vdash e(v_i) \ : \ \tau}{\Gamma \vdash v \leftarrow e \ : \ void} \ [ASSIGN]$$

$$binop \in \{+, -, *, /, **\}$$

$$\frac{Num\, a \quad \Gamma \vdash e_1 \ : \ a \quad \Gamma \vdash e_2 \ : \ a \quad binop \ : \ a \rightarrow a \rightarrow a}{\Gamma \vdash binop\ e_1 e_2 \ : \ a} \ [EXPR]$$

$$op \in \{+, -, *, /, **\}$$

$$Num\, \tau_1, \tau_2$$

$$\tau_1 <: \tau_2$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1\ \Gamma \vdash e_2 \ : \ \tau_2\ op \ : \ Num\, a \Rightarrow a \rightarrow a \rightarrow a}{\Gamma \vdash op\ cast\langle \tau_2 \rangle\, e_1\ e_2 \ : \ \tau_2} \ [BINOP]$$

$$relop \in \{<, \leq, =, \neq, >, \geq\}$$

$$Num\, \tau_1, \tau_2$$

$$\tau_1 <: \tau_2$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1\ \Gamma \vdash e_2 \ : \ \tau_2\ relop \ : \ Num\, a \Rightarrow a \rightarrow a \rightarrow Bool}{\Gamma \vdash op\ cast\langle \tau_2 \rangle\, e_1\ e_2 \ : \ Bool} \ [RELOP]$$

$$Num\, \tau_1, \tau_2$$

$$\tau_2 <: \tau_1$$

$$\frac{\Gamma \vdash v \ : \ \tau_1\ \Gamma(x_i) \vdash e(x_i) \ : \ \tau_2}{\Gamma \vdash v \leftarrow cast\langle \tau_1 \rangle\, e \ : \ void} \ [\text{ASSIGN SAFE CONV}]$$

$$Num\, \tau_1, \tau_2$$

$$\tau_1 <: \tau_2$$

$$\frac{\Gamma \vdash v \ : \ \tau_1\ \Gamma(x_i) \vdash e(x_i) \ : \ \tau_2}{\Gamma \vdash v \leftarrow cast\langle \tau_1 \rangle\, e \ : \ void} \quad [\text{ASSIGN UNSAFE CONV}]$$

Appendix 2: Run-time type checks

There are a few cases where type safety can't be guaranteed at compile time. For these cases, our compiler allows to insert run-time type checks.

A2.1: Run-time size checking for arrays slicing

If it is not possible to determine the size of the slice, our compiler will issue a type error. This can be relaxed to warning, in which case the compiler will insert a run-time check, for example:

Example 12 Run-time check for array slicing

```
integer a, s, i
dimension a(5,7), s(3)
! compute a, s, i
 if (size(s)==size(a(2,1:i:5)) then
   s = a(2,1:i:5)
 else
   print *, 'Type error: s and a(2,1:i:5) have a different SIZE'
   call exit(0)
 end if
```

In this way type, safety of non-constant array slices can be enforced at run time. Because the array slice assignment performs a data copy, the overhead of the if condition, which only performs a size check, is negligible.

A2.2: Run-time size checking for arrays as indices

If it is not possible to determine the size of the array, the compiler will issue a type error. This can be relaxed to warning, in which case the compiler will insert a run-time check. This is simply an if-then-else with the condition that `size(`*`lhs expr`*`) == size(`*`rhs expr`*`)`. For example:

Example 13 Run-time check for arrays as indices

```
integer a(5,5), b(3), k(5), i
! compute a, b, k, i
 if (size(b)==size(a(2,k(3:i)))) then
   b = a(2,k(3:i))
 else
   print *, 'Type error: b and a(2,k(3:i)) have a different SIZE'
   call exit(0)
 end if
```

In this way type, safety of non-constant array accesses can be enforced at run time. Because the use of arrays as indices results in a data copy, the overhead of the if condition, which only performs a size check, is negligible.

A2.3: Run-time checking of higher-order subroutines and functions

As subroutines and external functions can take the names of other subroutines or external functions as arguments, FORTRAN 77 has limited supports higher-order functions. The issue with the Fortran implementation of higher order functions is how they are typed. Recall that we can transform any function or subroutine into a pure function as discussed in Sect. 6.

Essentially, when a function $f_1(x)$ with type

$$f_1 : t_1 \rightarrow t_2$$

is passed as argument to a function $f_2(f)$, the type of f_2 is

$$f_2 : t_2 \rightarrow t_3$$

because in Fortran, only the return type of the passed function is used. The complete type would be

$$f_2 : (t_1 \rightarrow t_2) \rightarrow t_3$$

So in Fortran the type of the argument(s) of f_1 is not considered. Which means that code can easily be unsafe, as illustrated in Example 14.

Example 14 Unsafe code with higher-order function call

```
t3 function f2(f)
 t2 :: f,y
 external f
 t1 :: x
 t4 :: z
 logical :: c
 ...
 if (c) then
   y = f(x)
 else
   y = f(z,x)
 end if
end function
```

The call to *f(z,x)* in the example, while patently wrong, will pass silently. For subroutines the situation is the same, the only difference is that there is no return type so we use *void* as the type of the subroutine passed as argument.

Because the type information is incomplete, it is not possible to catch this type of error at compile time. However, we propose here a novel approach to detect the

behaviour at run time and throw an error. To do so, we construct a so-called *sum type* [6] at compile time. This is one kind of type used by many functional programming languages called algebraic data types (the other kind are *product types*, i.e. records), and it allows you to define a type with variants, so we can say "the type of this expression either be A or B or C, etc." . In particular, we use a sum type where every type variant is a function type, something like

$$datatype\ F = F_1\ t_1 \rightarrow t_2 \mid F_2\ t_4 \rightarrow t_1 \rightarrow t_2 \mid F_3\ \ldots$$

Then the type of the argument in the call becomes F but we can check which variant has been selected because effectively each argument is tagged with the name of the type variant. Fortran does not have such a type, but using an appropriate program transformation we can achieve the same effect. The approach is as follows:

- The types of all functions that are allowed to be passed as argument are known because these are the functions marked as EXTERNAL in the code unit containing the call (see Example 15)

Example 15 External

```
program functions_as_arguments
    external f3
   external f4
   t3 :: v1, v2, f2
   v1 = f2(f3,...)
   v2 = f2(f4,...)
  end
  t3 function f2(f,...)
   t2 :: f
   external f
   ...
  end
  t2 function f3(x)
   t1 :: x
   ...
  end
  t2 function f4(z,x)
   t1 :: x
   t4 :: z
   ...
  end
```

- Make a list of the type signatures of these functions:

$$F = [t_1 \rightarrow t_2, t_4 \rightarrow t_1 \rightarrow t_2, \ldots]$$

- The index in this list is a unique identifier for that type, *F(1)* refers to the first type, *F(2)* to the second, etc.
- We add the index variable as an additional argument to the calling function (Example 17) and use the index corresponding to the called function in the actual call (Example 16)

Example 16 Indexed external functions

```
program functions_as_arguments
 external f3 ! 1
 external f4 ! 2
 t3 :: v1, v2, f2
 v1 = f2(1,f3,...)
 v2 = f2(2,f4,...)
end
t3 function f2(idx,f,...)
 integer :: idx
 t2 :: f
 external f
 ...
end
t2 function f3(t1)
 t1 :: x
 ...
end
t2 function f4(t4,t1)
 t1 :: x
 t4 :: z
 ...
end
```

- We can now use this index to identify the selected type variant, as shown in Example 17. Every call to the function argument *f* is guarded by an if-then-else statement checking if the index *idx* matches the actual index corresponding to the variant. If this is not the case, a run-time type error is thrown. The variant is determined by matching the signature of the call to the signatures in the sum type.

Example 17 Identifying type variants using indexing

```
t3 function f2(idx,f)
 integer :: idx
 t2 :: f,y
 t1 :: x
```

```
t4 :: z
logical :: c
...
if (c) then
  if (idx==1) then
    y = f(x)
  else
    print *, 'Type error: call does not match signature'
    call exit(0)
  end if
else
  if (idx==2) then
    y = f(z,x)
  else
    print *, 'Type error: call does not match signature'
    call exit(0)
  end if
 end if
end function
```

This algorithm is formalised for subroutines as Algorithm 10; functions are entirely analogous. In this way, type safety of higher-order functions can be enforced at run time.

Algorithm 10: Type checking rules for variables in COMMON and EQUIVALENCE statements

This algorithm has linear complexity in the number of functions passed as arguments to a call. The run time overhead of the if condition is negligible for all but the most trivial function calls.

Acknowledgements The author acknowledges the support of the UK EPSRC under Grant EP/L00058X/1 and Dr. C. Brys for improving the structure and flow of the paper.

References

1. Abramson D, Lees M, Krzhizhanovskaya V, Dongarra J, Sloot PM, Yamamoto K, Uno A, Murai H, Tsukamoto T, Shoji F, Matsui S, Sekizawa R, Sueyasu F, Uchiyama H, Okamoto M, Ohgushi N, Takashina K, Wakabayashi D, Taguchi Y, Yokokawa M (2014) In: 2014 International conference on computational science the k computer operations: experiences and statistics. Procedia Computer Science, vol 29, pp 76–585
2. ANSI A (1978) Standard x3. 9-1978, programming language Fortran (revision of ansi x2. 9-1966). American National Standards Institute. Inc., NY, 197(8)
3. Colbrook A, Smythe C (1990) Formal specification of data abstraction in Fortran 77: abstract arrays. Softw Eng J 5(3):151–159. https://doi.org/10.1049/sej.1990.0017
4. Contrastin M, Rice A, Danish M, Orchard D (2016) Units-of-measure correctness in Fortran programs. Comput Sci Eng 18(1):102–107
5. Gordon S, McBride BJ (1994) Computer program for calculation of complex chemical equilibrium. NASA reference publication 1311
6. Hudak P, Hughes J, Peyton Jones S, Wadler P (2007) A history of Haskell: being lazy with class. In: Proceedings of the third ACM SIGPLAN conference on history of programming languages, pp 12–1–12–55
7. Kämpf J (2009) Ocean modelling for beginners: using open-source software. Springer, Berlin
8. Liao C, Quinlan DJ, Panas T, De Supinski BR (2010) A rose-based openmp 3.0 research compiler supporting multiple runtime libraries. In: International workshop on OpenMP. Springer, Berlin, pp 15–28
9. Maley D, Kilpatrick P, Schreiner E, Scott N, Diercksen G (1996) The formal specification of abstract data types and their implementation in Fortran 90: implementation issues concerning the use of pointers. Comput Phys Commun 98(1):167–180. https://doi.org/10.1016/0010-4655(96)00093-8
10. Milner R (1978) A theory of type polymorphism in programming. J Comput Syst Sci 17(3):348–375
11. Orchard D, Rice A (2013) Upgrading Fortran source code using automatic refactoring. In: Proceedings of the 2013 ACM workshop on refactoring tools, WRT'13. ACM, New York, NY, USA, pp 29–32. https://doi.org/10.1145/2541348.2541356
12. Overbey J, Xanthos S, Johnson R, Foote B (2005) Refactorings for fortran and high-performance computing. In: Proceedings of the second international workshop on Software engineering for high performance computing system applications. ACM, pp 37–39
13. Pierce BC, Benjamin C (2002) Types and programming languages. MIT Press, Cambridge
14. Reid J (2018) The new features of Fortran 2018. In: ACM SIGPLAN fortran forum, vol 37. ACM New York, NY, USA, pp 5–43
15. Reid N, Wray J (1999) A prescriptive semantics of fortran 95. In: ACM SIGPLAN fortran forum, vol 18. ACM New York, NY, USA, pp 2–3
16. Scott N, Kilpatrick P, Maley D (1994) The formal specification of abstract data types and their implementation in Fortran 90. Comput Phys Commun 84(1):201–225. https://doi.org/10.1016/0010-4655(94)90212-7
17. Takemi T, Yoshida T, Horiguchi M, Vanderbauwhede W (2020) Large-eddy-simulation analysis of airflows and strong wind hazards in urban areas. Urban Clim 32:100625
18. Tinetti FG, Méndez M (2012) Fortran legacy software: source code update and possible parallelisation issues. In: ACM SIGPLAN fortran forum, vol 31. ACM, pp 5–22
19. Vanderbauwhede W (2018) The glasgow Fortran source-to-source compiler. J Open Source Softw 3(32):865
20. Vanderbauwhede W, Davidson G (2018) Domain-specific acceleration and auto-parallelization of legacy scientific code in Fortran 77 using source-to-source compilation. Comput Fluids 173:1–5