The Journal of Open Source Software

fortran-src: Fortran static analysis infrastructure

Mistral Contrastin [1,2], **Raoul Hidalgo Charman** [3], **Matthew Danish** [4], **Benjamin Orchard** [5], **Dominic Orchard** [5,6], **Andrew Rice** [1,7], **and Jason Xu** [3]

1 Department of Computer Science and Technology, University of Cambridge **2** Meta **3** Bloomberg **4** Utrecht University **5** School of Computing, University of Kent **6** Institute of Computing for Climate Science, University of Cambridge **7** GitHub

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

DRAFT

Summary

fortran-src is an open source Haskell library and command-line application for the lexing, parsing, and static analysis of Fortran source code. It provides an essential front-end interface to build other Fortran language tools, e.g., tools for static analysis, automated refactoring, verification, and compilation. The tool provides multiple parsers which support Fortran source code conforming to the FORTRAN 66, FORTRAN 77, Fortran 90 and Fortran 95 standards, as well as some legacy extensions and partial Fortran 2003 support. The parsers generate a shared Abstract Syntax Tree representation (AST), over which a variety of core static analyses are defined to facilitate the development of analysis and language tools. The library has been deployed in a number of projects in both academia and industry to help build further language tools for manipulating Fortran.

Statement of need

As one of the oldest surviving programming languages (Backus, 1978), Fortran is used in a vast amount of software still in deployment. Fortran is not only a mainstay of legacy software, but is also still used to write new software, particularly in the sciences. Given the importance of numerical models in science, verifying the correctness of such models is critical for scientific integrity and progress. However, doing so is difficult, even more so than for traditional software; for computational models, the expected program behaviour is often unknown, uncertainty is the rule, and approximations are pervasive. Despite decades of progress in program verification within computer science, few formal verification techniques are currently applied in scientific software. To facilitate a step-change in the effectiveness of verification for computational science, a subset of the authors of this paper developed a suite of verification and static analysis tools named CamFort to explore lightweight verification methods (requiring little or no programmer effort), targeted at scientific programming (Contrastin et al., 2016). We chose Fortran as it remains a popular language in the international scientific community; Vanderbauwhede (2022) reports data from 2016 on the UK's "Archer'' supercomputer, showing the vast majority of use being Fortran code. Fortran is particularly notable for its prevalence in earth sciences, e.g., for implementing global climate models that then inform international policy decisions (Méndez et al., 2014). In 2024, Fortran re-entered the Top 10 programming languages in the TIOBE Index, showing its enduring popularity.

The continued use of Fortran, particualarly in scientific contexts, was the catalyst for the fortran-src software package.

One of the challenges in writing language tools for Fortran is its long history. There have been several major language standards (FORTRAN I-IV, FORTRAN 66 and 77, Fortran 90, 95, 2003, 2008, 2018 and more) or *restandardisations*. Newer standards often deprecate features which were known to be a ready source of error, or were difficult to specify or understand. However, compilers often support an amalgam of features across language standards, including deprecated features (Urma et al. (2014)). This enables developers to keep using deprecated features, or mix a variety of language standard styles. This complicates the task of developing new tools for manipulating Fortran source code; one must tame the weight of decades of language evolution.

This package, fortran-src, provides an open-source unified core for statically analysing Fortran code across a variety of standards, with a focus on legacy code over cutting-edge modern Fortran. It includes a suite of standard static analyses and tools to be used as a basis for further programming language tools and systems. It provides the core front-end and has been released as a standalone library and tool.

Related software

A variety of other tools exist for analysing Fortran, but those we have found are all commerical and closed source, e.g., plusFORT[1] (which includes the SPAG refactoring tool), the SimCon fpt tool[2] (which includes further verification features like dimensional analysis), and Forcheck[3]. General commerical static analysis tools, like Coverity[4] and Understand[5], can also handle Fortran. Photran[6] is an open source plugin for refactoring in Eclipse, but does not provide more general static analysis facilities. More recent work has developed open source tools for refactoring Fortran (Vanderbauwhede, 2022): RefactorF4Acc[7] is an open source tool for upgrading FORTRAN 77 code to Fortran 95.

Functionality

fortran-src provides the following functions over Fortran source code:

- lexing and parsing to an expressive abstract syntax tree;
- perform various static analyses;
- pretty printing;
- "reprinting", or patching sections of source code without removing secondary notation such as comments;
- exporting to JSON.

fortran-src is primarily a Haskell library, but it also packages a command-line tool for running and inspecting analyses. By exporting parsed code to JSON, the parsing and standard analyses that fortran-src provides may be utilized by non-Haskell tools.

The library's top-level module is called `Language.Fortran`. As such all submodules are within that namespace.

Lexing and parsing

Static analysis of Fortran requires a choice in the lexing and parsing front end: either to take the approach of many compilers, allowing an amalgam of features (e.g., gfortran with its hand-written parser), or to enforce language standards at the exclusion of some code that is accepted by major compilers. fortran-src takes roughly the latter approach, though it also has an extended Fortran 77 mode for supporting legacy extensions influenced by vendor-specific compilers that have been popular in the past.

[1] https://polyhedron.com/?product=plusfort
[2] http://simconglobal.com/fpt_summary.html
[3] https://codework.com/solutions/developer-tools/forcheck-fortran-analysis/
[4] https://www.synopsys.com/software-integrity/static-analysis-tools-sast/coverity.html
[5] https://scitools.com/
[6] https://projects.eclipse.org/projects/tools.ptp.photran
[7] https://github.com/wimvanderbauwhede/RefactorF4Acc

DRAFT

Furthermore, the Fortran language has evolved through two broad syntactic forms:

- *fixed source form*, used by FORTRAN 66 and FORTRAN 77 standards, where each line of source code follows a strict format (motivated by its original use with punched cards). The first 6 columns of a line are reserved for labels and continuation markers. The character `C` in column 1 indicates a comment line to be ignored by the compiler, else the line properly begins from column 7.
- *free source form*, first specified in Fortran 90 and subsequent versions of the standards, which has fewer restrictions on line format and a different method of encoding line continuations.

Therefore, two lexers are provided: the fixed form lexer, for handling earlier versions of the language: FORTRAN 66 and FORTRAN 77 (and additional `Legacy` and `Extended` modes), and the free form lexer, for Fortran 90 onwards. The lexers are auto-generated via the alex tool.

The fixed form lexer (`Language.Fortran.Parser.Fixed.Lexer`) handles the expectation that the first 6 columns of a line are reserved for code labels and continuation line markers, with code starting at column 7, and with comment lines starting with `C` in the first column. Only the first 72 columns are scanned (i.e., anything after is ignored).

The free form lexer (`Language.Fortran.Parser.Free.Lexer`) is less constrained but still has to manage continuation-line markers which break statements across multiple lines.

fortran-src then defines one parser per supported standard (with the exception of FORTRAN 77, for which we define extra parsers handling non-standard extended features). Each parser uses the source form that its standard specifies. Later Fortran standards such as Fortran 2003 are generally comparable to Fortran 90, but with additional syntactic constructs. The fortran-src parsers reflect this, gating certain features by the language standard being parsed. Parsers are grouped by fixed or free form, thus parsers for FORTRAN 66 and FORTRAN 77 are within the `Language.Fortran.Parser.Fixed` namespace and the rest are within `Language.Fortran.Parser.Free`. A top-level module (`Language.Fortran.Parser`) provides a unified point of access to the underlying parsers.

The suite of parsers is automatically generated from attribute grammar definitions in the Bison format, via the happy tool. CPP (the C pre-processor) can be run prior to lexing or parsing.

Unified Fortran AST

The parsers all share a common abstract syntax tree (AST) representation (`Language.Fortran.AST`) via a group of mutually-recursive data types. All such data types are *parametric data types*, parameterised by the type of "annotations" that can be stored in the nodes of the tree. For example, the top-level of the AST is the `ProgramFile a` type, which comprises a list of `ProgramUnit a` values, parameterised by the annotation type `a` (i.e., that is the generic type parameter). The annotation facility is useful for, for example, collecting information about types within the nodes of the tree, or flagging whether the particular node of the tree has been rewritten or refactored.

An interface of functions provides the ability to extract and set annotations via the `Annotated` class, of which all AST data types are an instance:

```
class Annotated f where
  getAnnotation    :: f a -> a
  setAnnotation    :: a -> f a -> f a
  modifyAnnotation :: (a -> a) -> f a -> f a
```

Some simple transformations are provided on ASTs:

- Grouping transformation, turning unstructured ASTs into structured ASTs (`Language.Fortran.Transformation.Grouping`);
- Disambiguation of array indexing vs. function calls (as they share the same syntax in Fortran) (`Language.Fortran.Transformation.Disambiguation`) and intrinsic calls from regular function calls (`Language.Fortran.Transformation.Disambiguation.Intrinsic`), e.g. `a(i)` is both the syntax for indexing array `a` at index `i` and for calling a function named `a` with argument `i`;
- Fresh name transformation (obeying scoping) (`Language.Fortran.Analysis.Renaming`).

All of these transformations are applied to the ASTs following parsing (with some slight permutations on the grouping transformations depending on whether the code is FORTRAN 66 or not).

Static analyses

The table below summarises the current static analysis techniques available within fortran-src (grouped under `Language.Fortran.Analysis`).

- Control-flow analysis (building a super graph) (`Language.Fortran.Analysis.BBlocks`);
- General data flow analyses (`Language.Fortran.Analysis.DataFlow`), including:
  - Reaching definitions;
  - Def-use/use-def;
  - Constant evaluation;
  - Constant propagation;
  - Live variable analysis;
  - Induction variable analysis.
- Type analysis (`Language.Fortran.Analysis.Types`);
- Module graph analysis (`Language.Fortran.Analysis.ModGraph`);

A representation, abstracted away from the details of the syntax tree, is provided for evaluation of expressions and for semantic analysis (`Language.Fortran.Repr`). Constant expression evaluation (`Language.Fortran.Repr.Eval.Value`) leverages this representation and enables some symbolic manipulation too, essentially providing some partial evaluation.

Pretty printing, reprinting, and rewriting

A commonly required feature of language tools is to generate source code. We thus provide pretty printing features to generate textual source code from the internal AST (`Language.Fortran.PrettyPrint`).

Furthermore, fortran-src provides a diff-like patching feature for (unparsed) Fortran source code that accounts for the fixed form style, handling the fixed form lexing of lines, and comments in its application of patches (`Language.Fortran.Rewriter`). This aids in the development of refactoring tools.

The associated `CamFort` package[8] which builds heavily on fortran-src provides a related "reprinting" algorithm (Clarke et al. (2017)) that fuses a depth-first traversal of the AST with a textual diff algorithm on the original source code. The reprinter is parameterised by `reprintings` which hook into each node and allow nodes which have been refactored by CamFort to have the pretty printer applied to them. The resulting outputs from each node are stitched into the position from which they originated in the input source file. This further enables the development of refactoring tools that need to perform transformations on source code text.

[8] https://github.com/camfort/camfort

Example usage

Example command-line tool use

The bundled executable fortran-src exposes tools for working with Fortran source code, including inspecting analysis results (such as the program basic blocks or inferred variable and function types) and code reformatting.

In the following examples, the file main.f90 is a Fortran 90-compatible program with the following content:

```
program main
    implicit none

    real :: r, area
    r = 1.0
    area = area_of_circle(r)
    print *, area

    contains

    function area_of_circle(r) result(area)
        real, parameter :: pi = 3.14
        real, intent(in) :: r
        real :: area
        area = r * r * pi
    end function
end program
```

DRAFT

The fortran-src binary must be on your system path in order to invoke it. Alternatively, if you use Stack to build the project, you may replace the fortran-src prefix with stack run -- and invoke it directly in the project directory.

Invocations follow the common syntax fortran-src <FILE> <OPTIONS>. You select the command you wish to run using the relevant option. Run fortran-src --help to view a built-in description of the options available.

The extension of the input file determines which Fortran version the file is parsed as. This may be overriden by explicitly requesting a specific version:

```
fortran-src main.f90 --fortranVersion=90 <COMMAND>
```

Running fortran-src with no arguments displays the included help, which includes an enumeration of the Fortran versions supported.

Parse a file and view the typechecker output: fortran-src main.f90 --typecheck

```
4:12     r               Real 4 Variable
4:15     area            Real 4 Variable
11:4     area_of_circle         - Function
12:27    pi              Real 4 Parameter
13:28    r               Real 4 Variable
14:16    area            Real 4 Variable
```

A file can be parsed and then pretty-printed back using the fortran-src printing algorithm: fortran-src main.f90 --reprint

```
program main
  implicit none
  real :: r, area
```

```
  r = 1.0e0
  area = area_of_circle(r)
  print *, area

  contains

  function area_of_circle(r) result(area)
    real, parameter :: pi = 3.14e0
    real, intent(in) :: r
    real :: area
    area = ((r * r) * pi)
  end function area_of_circle
end program main
```

Note the printing functionality has added the additional information of the program unit name on the end lines here.

Fortran code is printed with its corresponding *source form* i.e. FORTRAN 77 code is printed using fixed source form, while Fortran 90 and above use free source form.

Example library use: parsing and printing

The following simple Haskell example shows how to import the general parser module, fix the language version to Fortran 90, parse some code into the AST, and then print it to standard output:

```haskell
module Tmp where

import qualified Language.Fortran.Parser  as F.Parser
import qualified Language.Fortran.Version as F

import qualified Data.ByteString.Char8 as B

main :: IO ()
main = do
    v <- askFortranVersion
    let parse = F.Parser.byVer v
    case parse "<no file>" program of
      Left  err -> putStrLn $ "parse error: " <> show err
      Right ast -> print ast

askFortranVersion :: IO F.FortranVersion
askFortranVersion = return F.Fortran90

program :: B.ByteString
program = B.pack $ unlines $
  [ "function area_of_circle(r) result(area)"
  , "    real, parameter :: pi = 3.14"
  , "    real, intent(in) :: r"
  , "    real :: area"
  , "    area = r * r * pi"
  , "end function"
  , ""
  , "program main"
  , "    print *, area_of_circle(1.0)"
  , "end program"
```

DRAFT

JOSS The Journal of Open Source Software

```
]
```

A simple way of testing this example is to install the `fortran-src` package via cabal (i.e., `cabal install fortran-src`) to make it available within your environment for GHC.

Example library use: Analysis of balanced ALLOCATE statements

Let's say we wish to write a new Fortran code analysis using fortran-src. Fortran 90 introduced *allocatable arrays*, which enable declaring and using dynamic arrays in a straightforward manner. Allocatable arrays are declared only with the scalar type and rank, omitting the upper bound:

```fortran
integer, dimension(:), allocatable :: xs
```

A newly-declared allocatable begins *unallocated*. Reading from an unallocated array is an erroneous operation. You must first `allocate` the array with dimensions:

```fortran
! allocate memory for an array of 5 integers
! (note that the array is not initialized)
allocate(xs(5))
```

When finished, you must manually `deallocate` the array.

```fortran
deallocate(xs)
```

Arrays must be deallocated before they go out of scope, or else risk leaking memory. As an example use of fortran-src, we show here a simple code pass that asserts this property. Since arrays may be deallocated and re-allocated during their lifetime, we shall track the allocatables currently in scope, and assert that all are unallocated at the end of the program unit. Fortran being highly procedural means it lends itself to monadic program composition, so we first design a monad that supports tracking allocatables. (We use the `effectful` effect library here, but the details are insignificant). The full code listing is available online.[9]

```haskell
import qualified Language.Fortran.AST as F
-- ....
-- Declare an effectful interface for the static analysis
data Analysis :: Effect where
    DeclareVar         :: F.Name -> Analysis m ()

    MakeVarAllocatable :: F.Name -> Analysis m ()
    AllocVar           :: F.Name -> Analysis m ()
    DeallocVar         :: F.Name -> Analysis m ()

    AskVar             :: F.Name -> Analysis m (Maybe VarState)

    -- extra: enable emitting other semi-relevant analysis info
    EmitErr            :: String -> Analysis m a
    EmitWarn           :: String -> String -> Analysis m ()

-- Representation of variable information for the analysis
data VarState
  -- ¦ Declared.
  = VarIsFresh

  -- ¦ Allocatable. Counts number of times allocated.
  | VarIsAllocatable AllocState Int
    deriving stock Show
```

[9] https://github.com/camfort/allocate-analysis-example

```haskell
data AllocState
  = Allocd
  | Unallocd
    deriving stock Show
```

Now we can design a mini program in this monad by pattern matching on the Fortran AST data types from fortran-src:

```haskell
-- Analyse statements
analyseStmt :: Analysis :> es => F.Statement a -> Eff es ()
analyseStmt = \case
  -- Emit declarations
  F.StDeclaration _ _ _ attribs decls ->
    traverse_ (declare attribs) (F.aStrip decls)

  -- Emit allocatable names
  F.StAllocatable _ _ decls ->
    traverse_ makeAllocatable (F.aStrip decls)

  -- Emit allocated variables
  F.StAllocate   _ _ _ es _ ->
    traverse_ allocate (F.aStrip es)

  -- Emit deallocated variables
  F.StDeallocate _ _   es _ ->
    traverse_ deallocate (F.aStrip es)

  -- Check usage in any other statements
  st -> analyseStmtAccess st

-- Handle a declaration
declare
    :: Analysis :> es
    => Maybe (F.AList F.Attribute a) -> F.Declarator a -> Eff es ()
declare mAttribs d =
    case F.declaratorVariable d of
      F.ExpValue _ _ (F.ValVariable dv) -> do
        declareVar dv
        case mAttribs of
          Nothing      -> pure ()
          Just attribs ->
            if   attribListIncludesAllocatable (F.aStrip attribs)
            then makeVarAllocatable dv
            else pure ()
      _ -> emitWarn "bad declarator form" "ignoring"

-- Handle an allocation expression
allocate :: Analysis :> es => F.Expression a -> Eff es ()
allocate = \case
  F.ExpSubscript _ _ (F.ExpValue _ _ (F.ValVariable v)) _dims ->
    allocVar v
  _ -> emitWarn "unsupported ALLOCATE form" "ignoring"

-- Handle a deallocation expression
deallocate :: Analysis :> es => F.Expression a -> Eff es ()
deallocate = \case
```

DRAFT

```
  F.ExpValue _ _ (F.ValVariable v) ->
    deallocVar v
  _ -> emitWarn "unsupported DEALLOCATE form" "ignoring"
```

We wish to evaluate this mini program to receive a report of the allocatable variables and whether they were properly deallocated. A state monad holding a map of variable names to `VarState` entries can implement this, and we bolt this on top of `IO` for easy emission of warnings and errors. The `runAnalysis` function then handles the effect interface, using the stateful map to store information about our variables, i.e., whether they are allocatable, allocated, deallocated, or neither:

```
-- 'F.Name' is the type synonym for variable names
type Ctx = Map F.Name VarState

runAnalysis
    :: (IOE :> es, State Ctx :> es)
    => Eff (Analysis : es) a
    -> Eff es a
-- e.g. @'AskVar' v@ gets mapped to @'Map.lookup' v ctx@
```

For the sake of brevity, we include just the code for handling the deallocation operation. To handle deallocations, we look up the deallocated variable in the map and report on various behaviours that would be program errors in the Fortran code: (1) the deallocated variable does not exist; (2) the deallocated variable is not allocatable; (3) the deallocated variable is allocatable but has not been allocated.

Lastly (4) is a non-buggy situation where the deallocated variable is allocatable and is allocated, but is not marked as unallocated.

```
  DeallocVar v -> do
    st <- State.get
    case Map.lookup v st of
      -- (1) Variable was never declared
      Nothing  -> err "tried to deallocate undeclared var"

      -- ... variable is declared
      Just vst ->

        case vst of
      -- (2) Variable is not allocatable
          VarIsFresh -> err "tried to deallocate unallocatable var"

      -- ... variable is allocatable
          VarIsAllocatable vstAllocState vstAllocCount ->

        -- Check its state
        case vstAllocState of

          -- (3) Trying to deallocate unallocated variable
              Unallocd -> err "tried to deallocate unallocated var"

          -- (4) Deallocating allocated variable
              Allocd -> do
                let vst' = VarIsAllocatable Unallocd vstAllocCount
                State.put $ Map.insert v vst' st
```

Note, this analysis does not handle control flow operators. Further work may involve tracking allocatable status specially in data flow analyses.

DRAFT

Work building on fortran-src

CamFort

As mentioned in the introduction, the origin of fortran-src was in the CamFort project and its suite of tools. The aim of the CamFort project[10] was to develop practical tools for scientists to help reduce the accidental complexity of models through evolving a code base, as well as tools for automatically verifying that any maintenance/evolution activity preserves the model's behaviour. The work resulted in the CamFort verification tool for Fortran[11] of which fortran-src was the core infrastructure developed for the tool.

CamFort provides some facilities for automatically refactoring deprecated or dangerous programming patterns, with the goal of helping to meet core quality requirements, such as maintainability (D. Orchard & Rice (2013)). For example, it can rewrite EQUIVALENCE and COMMON blocks (both of which were deprecated in the Fortran 90 standard) into more modern Fortran style. These refactorings also help expose any programming bugs arising from bad programming practices.

The bulk of the features are however focussed on code analysis and lightweight verification (Contrastin et al. (2016)). Source-code annotations (comments) provide specifications of certain aspects of a program's meaning or behaviour. CamFort can then check that code conforms to these specifications. CamFort can also suggest places to insert specifications and, in some cases, infer the specifications of existing code. Facilities include: units-of-measure typing (D. Orchard et al. (2020),D. A. Orchard et al. (2015),Danish et al. (2024)), array access patterns (for capturing the shape of stencil computations that can be complex, involving intricate index manipulations) (D. Orchard et al. (2017)), deductive reasoning via pre- and post-conditions in Hoare logic style, and various code safety checks such as memory safety by ensuring every ALLOCATE has a DEALLOCATE, robustness by analysing the use of conditionals on floating-point numbers, and performance bug checks on arrays (e.g., that the order of array indexing using induction variables matches the order of enclosing loops defining those induction variables). CamFort has been previously deployed at the Met Office, with its analysing tooling run on the Unified Model (Walters et al. (2017)) to ensure internal code quality standards are met.

Further analyses building on fortran-src

fortran-vars memory model library

`fortran-vars` is a static analysis library built on top of `fortran-src`. Many static analysis questions depend on knowing the value and type of expressions. `fortran-vars` provides an API to answer this fundamental question. It has modules for symbol table construction, constant expression evaluation, and type checking. Additionally, `fortran-vars` provides a memory model to resolve aliases introduced by `equivalence` statements, which are very common in legacy Fortran 77 code. It is possible to construct such a memory model because variables in Fortran 77 are statically allocated by default. Data flow analysis, such as constant propagation analysis, can be conducted based on memory locations instead of variable names.

Nonstandard INTEGER refactoring

Outside of CamFort, fortran-src has been used to build other (closed source) refactoring tools to help migration and improve the quality of large legacy codebases, building on top of the library's AST, analysis, and reprinting features.

[10] Funded from 2015-18 by the EPSRC under the project title *CamFort: Automated evolution and verification of computational science models* https://gow.epsrc.ukri.org/NGBOViewGrant.aspx?GrantRef=EP/M026124/1

[11] https://github.com/camfort/camfort

The Journal of Open Source Software

DRAFT

One example of this has been an effort to fix a number of issues regarding the use of integers used where logical types are expected. There are four main issues with this:

- Use of integers with logical operators, which often behave as bitwise functions in a way that propagate up in compound logical expressions;
- Assignment of integers to logicals, some compilers 'normalizing' the value;
- Use of integers in conditionals, the evaluation of which varies by compiler;
- Use of arithmetic operators with logicals, which can have unexpected results when the underlying value is not a 0 or 1.

Combined, this can lead to some very unexpected behaviour, such as the following snippet:

```
      integer is_foo
      if (.not. is_foo()) then
c     important code to *not* run when foo is true
c     ...
      endif
```

Even if `is_foo` returns the expected values of 0 and 1, the `.not.` performs as a bitwise not, and then, if the compiler determines it to be true by being not equal to 0, then this will *always* return true.

In order to fix these cases, additional tooling was deployed to gather information about whether a given function returned a 0 or 1. With this, a tool was written to refactor many expressions by using the fortran-vars typechecker to find integer expressions and normalise them using `.ne. 0` while flagging anything potentially changing behaviour for further manual inspection. These might be situations in which some code is hard to statically analyse but safe, or it may have uncovered an existing bug. The tool uncovered many such bugs in a particular codebase during this effort, including several in the form of the snippet above.

This effort, along with a number of others, allowed the team working at Bloomberg (a subset of the authors here) to eventually migrate a codebase from a legacy compiler to a modified GFortran, with no change in behaviour. Ongoing efforts are using fortran-src to remove the patches on top of GFortran, as well as to introduce interfaces for more robust type checking in this code base.

Project maintenance and documentation

fortran-src may be built and used on Windows, Mac and Linux systems using a recent version of the Glasgow Haskell Compiler. The project includes an expansive test suite covering various parsing edge cases and behaviours, which is automatically executed for changes to the project (on the above three systems). Bug reports and other contributions are welcomed at the fortran-src GitHub page.

Acknowledgements

The initial work on the fortran-src infrastructure was funded by an EPSRC grant **CamFort: Automated evolution and verification of computational science models** (EP/M026124/1), from 2015-18, and by an EPSRC Impact Acceleration Award and then Knowledge Transfer Partnership grant from 2018-19. Orchard is also supported by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Sciences program, through which he carries on his work supporting scientists through programming languages, tools, and systems as part of the Institute of Computing for Climate Science at the University of Cambridge. Furthermore, this work was supported by a grant from Bloomberg.

A number of other people have been associated with the project and have contributed to the development of the package over the years (in alphabetical order of surname):

- Daniel Beer
- Anthony Burzillo
- Harry Clarke
- Aiden Jeffrey
- Lukasz Kolodziejczyk
- Vilem-Benjamin Liepelt
- Darius Makovsky
- Benjamin Moon
- Daniel Ruoso
- Eric Seidel
- Poppy Singleton-Hoare
- Jay Torry

References

Backus, J. (1978). The history of Fortran I, II, and III. *ACM Sigplan Notices*, *13*(8), 165–180.

Clarke, H., Liepelt, V.-B., & Orchard, D. (2017). Scrap Your Reprinter: A Datatype Generic Algorithm for Layout-Preserving Refactoring. *Pre-Proceedings of IFL 2017*.

Contrastin, M., Danish, M., Orchard, D., & Rice, A. (2016). Lightning talk: Supporting Software Sustainability with Lightweight Specifications. *Proceedings of the Fourth Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE4), University of Manchester, Manchester, UK, September 12-14*, *1686*.

Danish, M., Orchard, D., & Rice, A. (2024). *Incremental units-of-measure verification*. https://arxiv.org/abs/2406.02174

Méndez, M., Tinetti, F. G., & Overbey, J. L. (2014). Climate models: Challenges for fortran development tools. *2014 Second International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, 6–12.

Orchard, D. A., Rice, A. C., & Oshmyan, O. (2015). Evolving fortran types with inferred units-of-measure. *J. Comput. Science*, *9*, 156–162. https://doi.org/10.1016/j.jocs.2015.04.018

Orchard, D., Contrastin, M., Danish, M., & Rice, A. (2017). Verifying spatial properties of array computations. *Proceedings of the ACM on Programming Languages*, *1*(OOPSLA), 75.

Orchard, D., Contrastin, M., Danish, M., & Rice, A. (2020). Guiding user annotations for units-of-measure verification. *CoRR*, *abs/2011.06094*. https://arxiv.org/abs/2011.06094

Orchard, D., & Rice, A. (2013). Upgrading fortran source code using automatic refactoring. In E. R. Murphy-Hill & M. Schäfer (Eds.), *Proceedings of the 2013 ACM workshop on refactoring tools, WRT@SPLASH 2013, indianapolis, IN, USA, october 27, 2013* (pp. 29–32). ACM. https://doi.org/10.1145/2541348.2541356

Urma, R.-G., Orchard, D., & Mycroft, A. (2014). Programming language evolution workshop report. *Proceedings of the 1st Workshop on Programming Language Evolution*, 1–3. https://doi.org/10.1145/2717124.2717125

Vanderbauwhede, W. (2022). Making legacy fortran code type safe through automated program transformation. *The Journal of Supercomputing*, *78*(2), 2988–3028.

Walters, D., Boutle, I., Brooks, M., Melvin, T., Stratton, R., Vosper, S., Wells, H., Williams, K., Wood, N., Allen, T., & others. (2017). The Met Office unified model global atmosphere

DRAFT

6.0/6.1 and JULES global land 6.0/6.1 configurations. *Geoscientific Model Development*, *10*(4), 1487–1520.

DRAFT