

Global-Flow Compiler User Manual *

Department of Computer Science, Boston University

March 9, 2006

1 Usage

1.1 Installation

(build requirements, makefile instructions, etc.)

1.2 Command-line Invocation

(command line options)

1.3 Error Messages

(etc.)

2 Syntax

This section explains the syntax for the input file.

2.1 Convention

The input as a plain-text file in ASCII encoding. The file has a suffix “.gfs.” The input file is broken down to top-level sections or statements, namely:

- Specification - to declare a global-flow specification.
- Relations - to declare subtyping relation used to produce typing and type judgment.
- Localflows - to declare the type of localflow symbols.
- Typing - to produce typing of a given specification using a given relation and localflow definitions. Only a specification for which a typing is to be produced will be checked for correctness.
- Export - to specify which of the specifications, relations, localflows, and typing declarations to appear in the output object file.
- Import - to use the specification, relation, localflow, or typing, compiled in an object file, with the “.gfo” suffix.

A top-level statement generally produces “bindings” in a top-level dictionary that can be referred to by other statements when applicable. Inside the block of a statement, it may have its own name space that does not share the same name space with the top-level dictionary.

In the remaining subsections, we use the **typewriter** font to denote text in the input file. Furthermore, the **upright** font is used to denote the literal input string, but the *italicized* font is used to markup

*This work was supported in part by NSF grants ITR ANI-0205294, ANI-0095988, ANI-9986397, and EIA-0202067.

reference to syntax constructs and other information. In particular, `[` and `]` denote optional syntax, and `...` means repeating of some syntax.

There may be C-style comments in the input file, that is, a line that begins with `//` is a comment until the end of the line; a section that begins with `/*` until the closing `*/` marks a block of comment.

2.2 Specification

The specification section encloses the language of global flow expression as the following:

```
specification Symbol [free-variables] = begin
  expression
end
```

The *Symbol* is an identifier word that begins with a capital letter. This name is used as the key in the top-level dictionary name space to access the specification. The *free-variables* are optional entries in the form of (x, y, z, \dots) where x, y, z et al. are free variables of the *expression*. This can only be omitted when *expression* does not contain free variables. Technically, the compositional analysis engine automatically infers free variable occurrences, but in practice, we made it mandatory to avoid unwanted free variable leaking/capturing.

Our *expression* is the syntax of global-flow specification defined in the technical report *Safe Composition of Network Controllers*, but with laxed lexicographic conventions and the extension for `spec`. An identifier can be a word of multiple characters in length. An uppercase identifier, or a *Symbol*, is a word whose first letter is capitalized. A lowercase identifier, or a *var*, is a word whose first letter is not capitalized. The syntax can be any of the following:

- Lower case identifiers, *var*, denote flow variables.
- Upper case identifiers, *Symbol*, are local flows. This *Symbol* is defined in a `localflow` statement.
- “(*expression* ; *expression*)” is a sequential flow. Parentheses are not mandatory.
- “(*expression* || *expression*)” is a parallel flow. Parentheses are not mandatory. Space is not allowed between the two `|`’s.
- “let *var* = *expression* in *expression*” is a let-binding.
- “spec *Symbol*” uses the expression declared in a specification of the name *Symbol* in the place. However, this *Symbol* cannot recursively refer to the specification containing the expression itself. The reference is only resolved when the expression is checked with the `typing` statement.

Referring to another specification in the *expression* does not require that specification to be present, but a typing of it must be available. See the details on the `typing` statement.

2.3 Relations

The relation section effectively defines a subtyping relation for use in type checking a specification, i.e., Δ in *Safe Composition of Network Controllers*.

```
relations Symbol = begin
  BNF declarations
end
```

where the *BNF declarations* are written as one or more of the following (the declaration of `FwSocketType` and `BwSocketType` is mandatory):

- “`FwSocketType ::= form`” to declare the class of forward socket types.
- “`BwSocketType ::= form`” to declare the class of backward socket types.

- “*Symbol ::= form*” to declare a class named as *Symbol* that can be used in another *form*.

and the *form* can be one of the following:

- “*Symbol*” to refer to a previously defined class *Symbol*. This prevents the definition to become recursive.
- “(*form* [, *form* [...]])” is used to define a record of forms. A record type t_1 is a subtype of another record type t_2 if, when one lines up all members of t_1 and t_2 , all members of t_1 is a subtype of the corresponding member in t_2 .
- “!*form*” is to take the contravariant subtyping relation for *form*. That is, for all t_1, t_2 and a form f , $t_1 <:_f t_2$ iff $t_2 <:_{!f} t_1$.
- ““*string*₁” [| “*string*₂” [...]] where “*string*_{*i*}” <: “*string*_{*j*}” [; “*string*_{*i*}” <: “*string*_{*j*}” [...]]” defines a qualitative class with hard-coded relations. Note that reflexivity and transitivity are implied. Antisymmetry is checked by the parser.

There are also several predefined classes:

- **Numeric** is a class of all real numbers. The subtyping relation is less than or equal to (\leq).
- **Lexicographic** is a class of all “*string*” constants. The subtyping relation is less than or equal to, in lexicographic order comparison of strings.
- **Range** is a duple of numeric values denoting range. As a *form*, it may be defined as (! Numeric, Numeric).
- **Set** is a class whose subtyping relation is defined as the subset relation (\subseteq). The members of a set can be an arbitrary type. The members are unordered.

Note that *form* uniquely fixes the syntax for *socket-type*, as described in the next section.

2.4 Localflows

The top-level statement *localflows* defines a mapping of local- flow (as in the *expression* of a specification) to a flow-type.

```
localflows SymbolL: SymbolR = begin
  let Symbol = flow-type
  ...
end
```

where *Symbol*_L is the name of the localflow mapping, and *Symbol*_R is the corresponding subtyping relation (defined by a *relations* statement) that this mapping pertains to, since the forms in the subtyping relation uniquely determine the possible types.

A *flow-type* is a compound of four types: forward input, forward output, backward output, and backward input. Each of the types can be written according to the syntax of types in *Safe Composition of Network Controllers*:

- “*socket-type*” describes a socket type.
- “(*type* * *type*)” is a paired bundle of types. This is a type that naturally results from putting two global-flows in parallel. However, this is not the same as the record form of *socket-type*. The parentheses are mandatory.

and a *socket-type* can be any of the following:

- An arbitrary number in the IEEE floating point format. It can be a negative value and exponentiated, e.g., -3.718e10. This corresponds to the **Numeric** built-in class. Note that comparison is subject to numeric precision on the host computer.

- A string constant "*string*". This may be used with the `Lexicographic` class or a qualitative class.
- “(*socket-type* [, *socket-type* [...]])” is a record of *socket-types*, corresponding to the *form* of records. Each *socket-type* must match the sub-form declared in the record.
- “{ *socket-type* [, *socket-type* [...]] }” denotes a set of *socket-types*, which is used with the `Set` class. The members of the set are not restricted to any *form*.

Therefore, we write *flow-type* as “[*type*₁ , *type*₂ ; *type*₃ , *type*₄]”, where *type*₁ and *type*₂ are forward input and output types, and *type*₃ and *type*₄ are backward *output* and input types. Line breaks and spaces may be used anywhere to increase readability.

A localflows mapping is only associated with a specification when the typing judgment is carried out.

2.5 Typing

The `typing` statement tells the compiler to check a specification using a subtyping relation and a localflow definition.

```
typing SymbolT = check SymbolS : SymbolR [ using SymbolL ]
```

As mentioned earlier, an *expression* of form “spec *Symbol*_{S’}” as a reference to another specification is only resolved at this stage. When the compiler is asked to check for specification *Symbol*_S that contains a reference to *Symbol*_{S’}, the compiler looks up the typing for *Symbol*_{S’} that is checked against the same relation *Symbol*_R. Therefore, the specification for *Symbol*_{S’} does not have to be present, but the typing of it must either be in the same input file or imported.

If a typing for *Symbol*_{S’} is not found but the specification is, then an anonymous typing will be generated for this compilation session but not exported.

The user should be concerned if the typing for *Symbol*_{S’} is checked against the same localflows *Symbol*_L. The reason for this omission is because the precise definition of localflows might not be generally available.

Note that assigning a name to a typing does not affect the aforementioned look-up process. A name is used to specify what is to be exported, as seen in the next section.

2.6 Export

The `export` statement specifies which top-level *Symbols* are to be exported. These symbols refer to a specification, a relations, a localflow mapping, or a typing. Only those *Symbol* specified will be written to the output object file.

```
export Symbol [ Symbol [...] ]
```

2.7 Import

The `import` statement tells the global-flow compiler to load an object file and use the specification, relations, localflows, and typings stored in the object file.

```
import "string"
```

The "*string*" is a filename on the file-system. It is recommended that one omits the “.gfo” suffix in the filename, and put only the basename here. This file will be searched in a list of paths specified to the compiler.

2.8 Example

Here is an example “.gfs” input file illustrating the syntax.

```

/* filename: example.gfs */

/* description of a system with "holes," i.e., free variable occurrences. */

specification S0 (x, y, z) = begin
  (x || y); (y || z); (z || x)
end

/* description of a system that uses S0 while filling the "holes." */

specification S = begin
  let x = A in
  let y = B in
  let z = C in
  spec S0
end

/* want to check property based on range. */

relations D = begin
  FwSocketType ::= Range
  BwSocketType ::= "local" | "remote" | "any" | "both"
  where // in addition to reflexivity
    "both" <: "local"; "both" <: "remote"; "both" <: "any";
    "local" <: "any"; "remote" <: "any"; "both" <: "any"
end

/* localflow setting is always fixed on a relation */

localflows L: D = begin
  let A = [ (1.0, 2.0) , (0.75, 2.25) ;
    "remote" , "local" ]
  let B = [ (0.5, 2.5) , (1.0, 2.0) ;
    "both" , "local" ]
  let C = [ (0.75, 2.25) , (1.25, 1.75) ;
    "local" , "any" ]
end

typing T0 = check S0 : D using L

/* since specification S0 is used in S, this will automatically recall the
   typing T0 checked using the same L here. */

typing T = check S : D using L

/* write the relations, localflows, and the typings to example.gfo */

export D L T0 T

```

2.9 Formal Language

As a summary, we present the syntax of the input file in the style of context-free grammar with extensions from regular expression.

```

Top ::= (SpecStmt | RelStmt | LocalStmt | TypStmt | ExportStmt | ImportStmt)* EOF

SpecStmt ::= specification Symbol FreeVarDeclopt = begin Expr end
FreeVarDecl ::= () | (Var (, Var)* )
Expr ::= (Expr) | Symbol | Var | Expr ; Expr | Expr || Expr |
        let Var = Expr in Expr | spec Symbol

RelStmt ::= relations Symbol = begin BnfDecl* end
BnfDecl ::= Symbol ::= Form
Form ::= Symbol | (Form (, Form)* ) | !Form |
        String (| String)* where (String <: String)*

LocalStmt ::= localflows Symbol : Symbol = begin LocalDefn* end
LocalDefn ::= let Symbol = FlowType
FlowType ::= [ PlainType , PlainType ; PlainType , PlainType ]
PlainType ::= ( PlainType * PlainType ) | SocketType
SocketType ::= IEEEFloat | String | (SocketType (, SocketType)* ) |
        {SocketType (, SocketType)* }

TypStmt ::= typing Symbol = check Symbol : Symbol (using Symbol)opt

ExportStmt ::= export Symbol+

ImportStmt ::= import String

```

with the following terminals:

```

Symbol ::= upper-case identifiers
Var ::= lower-case identifiers
IEEEFloat ::= numerical values in IEEE floating point notation
String ::= any string surrounded by a pair of double-quotes
EOF ::= end of file

```

and the following reserved keywords and delimiters:

- Lower-case keywords: `export`, `import`, `specification`, `relations`, `localflows`, `typing`, `begin`, `end`, `let`, `in`, `spec`, `where`, `check`, `using`.
- Delimiters: `“;”`, `“|”`, `“|”`, `“=”`, `“: :=”`, `“<:”`, `“:”`, `“!”`, `“*”`, `“,”`, `“(”`, `“)”`, `“[”`, `“]”`, `“{”`, `“}”`.