

# The F7 Typechecker Manual

September 22, 2008

## Abstract

We describe an extended typechecker for F#, named F7. Its type system features refinement types, a form of dependent types that can be used to logically express and verify program specifications, with applications to security. We document our prototype implementation, describe current limitations, and discuss programming idioms.

We refer to Bengtson et al. [2008] for a general presentation of the type system and a series of programming examples.

## 1 Introduction

The F7 typechecker enables programmers to express and statically check *refinement types* for programs written in F#. The current implementation of the typechecker supports only a subset of F# and works best when type annotations are provided according to certain idioms. This document aims to define the programming language (and types) supported by the typechecker and to list some idioms that we have found useful in our case studies.

To use the typechecker, the programmer must define F7 interfaces for each F# module in the program. F7 interfaces have the suffix `.fs7` to distinguish them from normal F# interfaces (that have suffix `.fsi`); they may contain types that are outside the F# type system. F# modules have suffix `.fs` and contain unmodified F# code. In our current approach, all refinement type annotations must be given in the F7 interface.

Given an F# module `M.fs` with an F7 interface `M.fs7`, the typechecker can be invoked in two ways. First, it can be invoked with the `-genfsi` flag to generate an F# interface `M.fsi` by erasing all refinements:

```
f7.exe -genfsi <libraries> M.fs7
```

Second, it can be invoked to typecheck `M.fs` against the types declared in `M.fs7`.

```
f7.exe M.fs7 <libraries> M.fs
```

In general, these command lines would contain a series of F# modules and F7 interfaces, including interfaces for all the library modules. A special library interface, called `pervasives.fs7`, declares common F# types, such as integers, strings, booleans, and lists, and their common operators, such as arithmetic operations and logical connectives. This library can be extended to include any F# operator that the program needs to use.

## 2 Installation

The typechecker depends on the following software:

1. The .NET Framework, version 2.0 or higher.
2. The F# compiler, version 1.9.6.2  
(September 2008 CTP, <http://research.microsoft.com/fsharp/release.aspx>).
3. The Z3 SMT solver, version 1.3.6  
(<http://research.microsoft.com/projects/Z3/download.html>).

It has been tested for these versions on Windows Vista. The file README.txt provides detailed installation instructions.

## 3 Syntax of F7 Interfaces: .fs7 files

**Lexical Tokens** Variables (denoted by  $x_i$ ) begin with a upper- or lower-case letter followed by any combination of numbers, letters, primes ( $'$ ), and underscores ( $_$ ). Type variables (denoted by  $'a_i$ ) have a prime prefix. String, integer, and boolean literals are written as in F#.

### Types

$T ::=$	<code>unit,string,int,bool,list</code>	Predefined Types
	<code>'a</code>	Type Variable
	<code>(T1,...,Tn;M1,...,Mm) F</code>	Type Application (to type and value parameters)
	<code>x:T1 -&gt; T2</code>	Function Type (optional x has scope T2)
	<code>x1:T1 * ... * xn:Tn</code>	Tuple Type (optional $x_i$ has scope $T_{i+1}, \dots, T_n$ )
	<code>x:T{C}</code>	Refinement Type (optional x has scope C)

### Values and Formulas

$v,h,f ::=$	<code>x</code>	Local Variable
	<code>v.x</code>	Qualified Variable (x within module v)
$M ::=$	<code>qx</code>	Qualified Variable
	<code>c</code>	F# Constant: string, integer, bool
	<code>[]</code>	Empty List
	<code>M1::M2</code>	List Cons
	<code>(M1,...,Mn)</code>	Tuple
	<code>h(M1,...Mn)</code>	Constructor Application
	<code>{f1=M1;...fk=Mk}</code>	Record
	<code>M.fi</code>	Record Lookup
$C ::=$	<code>!x1,...,xk. C</code>	Universal Quantification ( $x_i$ has scope C)
	<code>?x1,...,xk. C</code>	Existential Quantification ( $x_i$ has scope C)
	<code>C1 =&gt; C2</code>	Implication

<code>C1 &lt;=&gt; C2</code>	Equivalence
<code>C1 \\/ C2</code>	Disjunction
<code>C1 /\ C2</code>	Conjunction
<code>not C</code>	Negation
<code>M1 = M2</code>	Equality
<code>M1 &lt;&gt; M2</code>	Inequality
<code>true, false</code>	Boolean Constants
<code>h(M1,...,Mn)</code> (C)	Predicate Application (h must be a constructor)

### Type Expressions and Type Definitions

<code>TE ::= ('a1,...'an; x1:T1,...,xm:Tm) F</code>	Abstract Type
<code>('a1,...'an; x1:T1,...,xm:Tm) F = T</code>	Type Abbreviation
<code>('a1,...'an; x1:T1,...,xm:Tm) F = {f1:T1; ... ;fk:Tk}</code>	Record Type
<code>('a1,...'an; x1:T1,...,xm:Tm) F = h1 of T1  ...  hk of Tk</code>	Algebraic Type
<code>TD ::= type TE1 and ... and TEn</code>	Mutually Recursive Types

Note: All record fields in a module must be distinct (as in OCaml)

### Interface Declarations

<code>param ::= public</code>	Export value/type in .fsi file
<code>private</code>	Only use for typechecking, do not export.
<code>&lt;empty&gt;</code>	Default: public
<code>D ::= param val x: T</code>	Value Declaration
<code>assume C</code>	Assumption
<code>ask C</code>	Query
<code>param TD</code>	Type Declaration
<code>exception h of T</code>	Exception Declaration
<code>open v</code>	Open Module Named v

`FS7 ::= D1 ... Dn`

## 4 Syntax of F# Modules: .fs files

### Constants

<code>Const ::= ()</code>	Unit
<code>true, false</code>	Booleans
<code>i</code>	Integers
<code>s</code>	Strings

## Patterns

P ::=	Const	Constant Pattern
	-	Wildcard
	x	Variable (binds x)
	h(P1,...,Pn)	Constructor Application
	P as x	As-pattern (binds x)
	(P1,...,Pn)	Tuple
	{f1=P1;...;fk=Pk}	Record

## Expressions

E ::=	Const	Constant
	v	Variable
	(E1,...,En)	Tuple
	{f1=E1;...;fk=Ek}	Record
	E.f	Record Lookup
	{E with f1=E1;...;fk=Ek}	Record Update
	fun x -> E	Anonymous Function (lambda expression)
	E1 E2	Function Application
	if E1 then E2 else E3	Conditional Branching
	let x:U = E1 in E2	Local Value Definition (type annotation U optional)
	let x P1 ... Pn = E1 in E2	Local Function Definition (defines x)
	let P = E1 in E2	Let Pattern (defines variables in P)
	E1 ; E2	Sequential Composition
	match E with	Pattern Matching
	P1 when E1 -> E1'	(when clause optional)
	...	
	Pk when Ek -> Ek'	
	function	Local Function with Pattern Matching
	P1 when E1 -> E1'	(when clause optional)
	...	
	Pk when Ek -> Ek'	
	raise h(E1,...En)	Exception
	failwith "..."	Exception
	Pi.assume (h(x1,...,xn))	Assume Fact
	Pi.expect (h(x1,...,xn))	Expect Fact

**F# Type Definitions** All types used in the F# module must be defined in the F7 interface. However, for F# compilation these types must also be defined within the F# file. We advocate that the programmer first write the F7 interface, then generate the F# interface, and then cut-and-paste the type definitions from the F# interface into the F# module. For reference, we define the supported subset of the F# type system here.

U ::= unit,string,int,bool,list   Predefined Types  
      'a                            Type Variable

<code>(U1,...,Un) F</code>	Type Application (only to type parameters)
<code>U1 -&gt; U2</code>	Function Type (no binder)
<code>U1 * ... * Un</code>	Tuple Type (no binder)
<code>FTE ::= ('a1,...'an) F = U</code>	Type Abbreviation
<code>('a1,...'an) F = {f1:U1; ... ;fk:Uk}</code>	Record Type
<code>('a1,...'an) F = h1 of U1   ...   hk of Uk</code>	Algebraic Type
<code>FTD ::= type FTE1 and ... FTEn</code>	Mutually-recursive Type Definition

### Module Definitions

<code>Def ::=</code>	<code>open v</code>	Open Module v
	<code>FTD</code>	Type Definition
	<code>exception h of T</code>	Exception Declaration
	<code>let x:U = E</code>	Module Value Definition (annotation U optional)
	<code>let P = E</code>	Module Pattern (defines variables in P)
	<code>let x P1 ... Pn = E</code>	Module Function Definition (defines x)
	<code>let rec x1 P11 ... P1n = E1</code>	Mutually-recursive Functions Definition
	<code>...</code>	
	<code>and xm Pm1 ... Pmn = Em</code>	
	<code>do E</code>	Evaluate Expression
<code>FS ::=</code>	<code>Def1 ... Defn</code>	F# Module

F# type annotations may be sprinkled around the program, for documentation, and to help the F# typechecker, but they are ignored by the F7 typechecker. The only type annotations it uses are those on let-expressions.

## 5 Command Line Arguments: f7.exe

The usage message for the F7 executable is as follows

F7 Typechecker as of 18 September 2008

```
Usage: f7.exe [--define flag] [-pervasives file] [-genfsi] [-genlst]
[-scripts file] [-debug] [-nokindcheck] [-timeout seconds] <file_1> ... <file_n>
```

```
--define <string>: Conditional Compilation Flag
-debug: Debug mode
-genfsi: Don't typecheck, only generate .fsi
-ask: Don't typecheck, but check logical queries in .fs7
-genlst: Generate lstlisting summary
-nokindcheck: Don't check kinds; assume partial trust
-timeout <int>: Timeout for Z3 process
-pervasives <string>: FS7 file for pervasives
-scripts <string>: Output file prefix for Z3 scripts
```

```
--help: display this list of options
-help: display this list of options
```

The filename arguments are interpreted according to their suffixes. If a file has a suffix `.fs7`, `.fsi`, `.ml7`, or `.mli` then the file is interpreted as an F7 interface. If it has a suffix `.fs` or `.ml`, it is interpreted as an F# module. All other files are ignored.

Typical usages can be found in the examples directory.

## 6 Known Issues

### 6.1 F# errors

Syntax errors in the F# code always result in errors in F7. Consider for instance this definition, with missing parentheses around `FooBar(x)`:

```
let fooBar = fun x -> expect FooBar(x); "bar"
```

This is a syntax error in F# and results in the following output from F7

```
a.fs(3,19): error FS0191: Successive arguments should be separated by spaces
or tupled, and arguments involving function or method applications should be
parenthesized.
```

We recommend that you always run the F# compiler first on your code, and fix all syntax errors and basic type errors before trying to typecheck through F7. The F# compiler typically gives better error messages and has better support for debugging.

### 6.2 Record field labels must be distinct

F7 does not allow the same field label to appear in different record types in the same module, even though F# does allow these. Say you declare two record types with the same label.

```
type t = {a:int; b:bool}
type r = {a:int; c:int}
```

Then you will see the following error message:

```
Failure: Microsoft.FSharp.Core.FailureException: Type error:
Duplicate definition of label a
at Typecheck.tyerr[T](Exception e)
```

### 6.3 No tuple-matching at top-level

Tuple-matching (and other pattern matching) is not allowed at the top-level; for example,

```
let x,y=(2,3)
```

yields something like

```
Failure: Microsoft.FSharp.Core.FailureException: File a.fs(3,5)-(3,8): error:
Patterns at top-level are unsupported at Typechecker.errorPos[T](...)
```

As a workaround, one can write:

```
let xy = (2,3)
let x = fst(xy)
let y = snd(xy)
```

## 6.4 Obscure error when forgetting to declare a function in the fs7 file

When you define a function in the fs file, but forget to declare it in the fs7 file, F7 complains that:

```
Typechecker+Cannot_gen_typeException:
Exception of type 'Typechecker+Cannot_gen_typeException' was thrown.
```

The fix is to declare your function in the fs7 file.

## 6.5 Discrepancy between fs7 and fs types

When there is a discrepancy between the fs7 and fs types, one may see something like the following, which indicates the location of a type error, though not exactly what has gone wrong. (This is reported as an `F#` error.)

```
File set-example.fs(44,2)-(44,3): error: This expression has type
```

## 6.6 Off-by-one error in smp file

When running Z3 directly on an smp file produced by F7, the queries are numbered from 1, while the comments within the smp file are numbered from 0.

## 6.7 Z3 not told that literal strings are distinct

If it matters, you can add assumptions like:

```
assume (not ("Bill" = "Steve"))
```

## 6.8 Nested comments are not allowed in an .fs7 file

```
(* foo (* bar *) baz *)
```

Fails with "Syntax error near line 2, character 20 in whatever.fs7"

## 6.9 Assume and Expect with non-atomic formulas

There is currently no support for general formulas in F#, so one needs to declare them using an auxiliary predicate. Hence, `Pi.assume` and `Pi.expect` only accept predicate applications, not arbitrary formulas or expressions.

For instance, `Pi.assume !x.CanRead(x,file)` does not compile in F#, but it may be coded as `Pi.assume AnyoneCanRead(file)` with additional F7 declarations

```
type predicate = ... | AnyoneCanRead of string
policy !file.( AnyoneCanRead(file) <=> !x.CanRead(x,file))
```

## References

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffei. "Refinement types for secure implementations". Technical Report MSR-TR-2008-118, Microsoft Research, September 2008.