Infrastructure for Analysis of F# Programs

Péter Diviánszky, Zoltán Horváth, Mónika Mészáros, Gábor Páli
divip@aszt.inf.elte.hu,{hz,bonnie,pgj}@inf.elte.hu
Department of Programming Languages and Compilers

Artyom Antyipin, Dorián Batha, Andrea Kovács, Péter Kovács, Judit Kőszegi, Dániel Leskó
Master students in Software Technology

Eötvös Loránd University, Budapest, Hungary

Introduction

Functional programming is an emerging programming paradigm. One of the recent languages is F#, a non-pure functional language. It has a growing importance mainly on .NET architecture.

F# was developed as a pragmatically-oriented variant of ML that shares a core language with OCaml. It executes at or near the speed of C# and C++, making use of the performance that comes through strong typing. F# includes extensions for working across languages, and it works seamlessly with other .NET programming languages and tools.

The purpose of this project is to develop tools enhancing programmer's efficiency in F#.

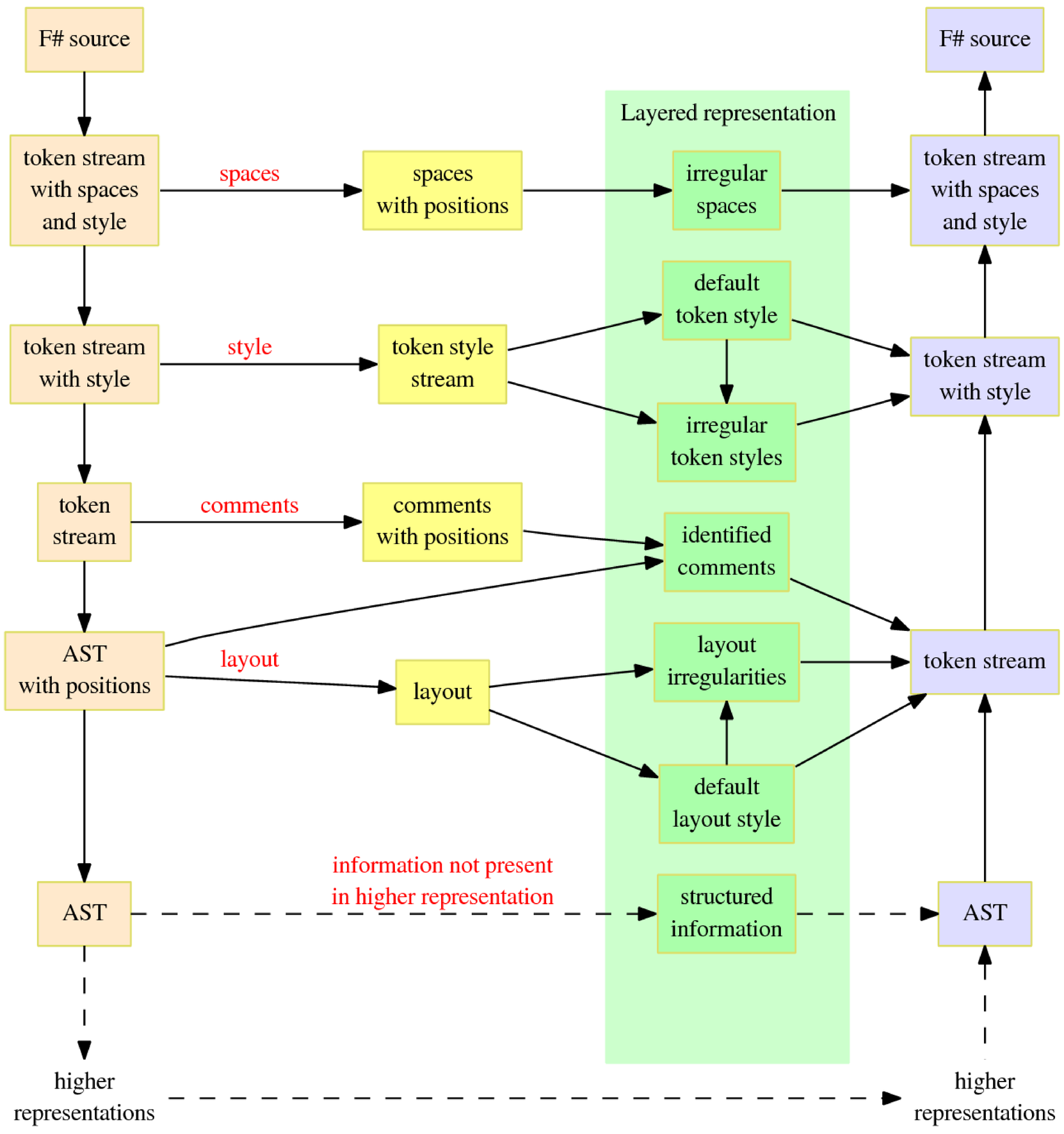
Separating Layers of Information

We present here an approach to analysis and refactoring programs, where separation of style of source code from content plays an important role. Our proposal resembles to the relation between HTML and CSS, and it could be adapted to arbitrary programming language.

Every information in the source code that does not change the effect of the resulting program code is considered style, for example line breaks, indentation, usage of comments and variable names, etc. Style of source code has great importance in human communication, for example, choice and use of a common style is essential for productivity in larger software projects.

Content and style can be seen as two layers of information. This dual model was refined to have more layers as shown in the graph below. Layers are constructed at every step, when the program is transformed to a higher representation. The resulted layer contains information not present in the higher representation. Hence layers do not overlap, e.g. do not contain redundant information.

We have developed an implementation in F#, where we would like to demonstrate the potential of this approach. Currently the layers regarding style are worked out and the elaboration of the higher-level layers (regarding content) is future work.



Software Based on the Concepts

In this infrastructure, program analysis (software metrics and resource analysis) can be done on the most suitable layer.

Refactoring and program transformation can be performed on the components of the layered representation which is the green area in the graph. In our opinion, a well-designed program transformation works with only one component. In consequence, a complex refactoring step should not bother with style information, which will be automagically adjusted to the transformed code (new and altered source segments will have default style which has been extracted from the program itself).

Concrete and nearly implemented applications:

- Analysis of style of programming, which is just a special kind of program analysis.

- Automatic unification of source code style, which is a refactoring step on the style component. For example, this application can be used for enforcing different coding policies.

```
let rec length l =
  match l with
      | []    ->  0
      | x::xs ->  1 + length xs
```

```
MatchStyle
 { rules  = InNewLine (Uniform (IndentedBy 4))
 , arrows = Uniform (InTheSameColunm 1)
 , expressions = Uniform (Floating 2)
 }
```

```
let rec length l =
  match l with
  | [] ->   0
  | x::xs-> 1 + length xs
```

```
MatchStyle
 { rules  = InNewLine (Uniform (IndentedBy 0))
 , arrows = NonUniform ( Floating 1
                       , [Index 2, Floating 0])
 , expressions = Uniform (InTheSameColunm 1)
 }
```

```
let rec length l =
  match l with
      | []           ->  0
      | x::longName ->  1 + length longName
```

The Model in Action

A small piece of the model is illustrated by three source code and two layout segments.

Source codes overlap their layout. Only the layouts of match expressions are shown. Unique identifiers link layout data to other data like the AST. (Eventually these layout fractions contain no identifiers.)

Layout and other data structured are designed so that it fits to program transformation. Consider the rename of the `xs` identifier in the upper code. This transformation does not affect style, but the synthesized code looks nice.

Supported by ELTE IKKK, Morgan Stanley