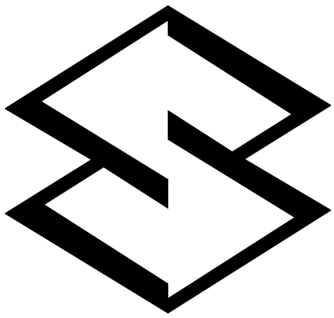
Supervision—F# for Static Analysis

Christian Clausen Bent Rasmussen

2012-08-29



Abstract

It is often clamied that functional languages are ideal for building static analysis applications. However, most applications are on small academic languages. In this document we test the claim on a real-world language, C/AL, which is the language used for defining business logic in Microsoft Dynamics NAV, a successful Enterprise Ressource Planning system.

We have used Microsoft's F# to build our application. F# is a multi-paradigm language, but we have almost exclusively used the functional aspects of the language.

Contents

Glossary

C/AL Common Application Language. The programming language used by the C/SIDE development environment. 4–6, 12–15

C/SIDE Client/Server Integrated Development Environment. 4, 5

CRM Customer Relationship Management. 4

ERP Enterprise Resource Planning. Business software platform serving a diverse set of needs–CRM being one of them. 4, 5

FOB File Binary container for C/AL objects. It may contain one or more objects. The FOB File format is more reliable than the Object Text File format but is harder to work with. In particular it does not have an open specification or open-source software to work with files in this format. Even third-party proprietary software for working with the FOB File format is close to non-existant. 4

IDE Integrated Development Environment. Software development software. Often with facilities for evelopment. 4

Microsoft Dynamics NAV Microsoft Enterprise Resource Planning (ERP) software. 4–6

Object A class-like program construct but without inheritance. This is the reason why C/SIDE is called object-based and not object-oriented. An object in the context of Dynamics NAV can mean either an object of a specific type or the use of an object of such a specific type. There are several basic types of objects in Dynamics NAV: table, page, report, codeunit, etc. 4

Object Text File Textual container for C/AL objects. It may contain one or more objects. The Object Text File format is more easy to work with than the FOB File format but is less reliable. 4

RAD Rapid Application Prototyping. Typically associated with a GUI interface for creating GUI interfaces with little or no programming needed, until events such as button clicks need to be handled. 4

1 Introduction

Microsoft Dynamics NAV is an ERP software platform. Developers write software for this platform via the C/SIDE development environment. The programming language for the environment is called C/AL.

The development environment that comes with Microsoft Dynamics NAV, C/SIDE, is pretty basic compared to development environments of more modern platforms like .NET and Java. Most strikingly, the C/SIDE development environment lacks an intuitive code browser that allows developers to browse code and find usages of variables and functions. Only very recently, support for "go to declaration" has been added to the C/AL editor, which is part of the C/SIDE development environment.

Additionally, the development environment does not include any tools that can analyse C/AL code statically, so it is not possible to run checks against coding conventions as it can be done with e.g. Java's Checkstyle [1] and FindBugs [2] or C#'s StyleCop [3], just to mention a few very popular tools.

As we both share an interest in static analysis and have worked with C/AL over the last 3-5 years, we decided to go on a mission to produce a suite of tools that improve developer productivity and enhances code quality.

This document focusses primarily on our experiences with F#. Prior to this project, we did not have any experience with F# at all, and only used functional languages for small experiments and a few "toy" projects during the study time.

2 Supervision

This section gives a short presentation of our first tool, Supervision. You can read more on our web-site, http://www.stati-cal.com/.

2.1 Benefits

The purpose of Supervisionis to give Microsoft Dynamics NAV [4] consultants, developers, and architects an easy and intuitive way to browse C/AL [5] source code, providing them with a better insight into the semantics of C/AL programs. This is useful when trying to understand how existing code works—for example before starting to customise the code—but also when performing (peer) review of new code.

2.2 Features

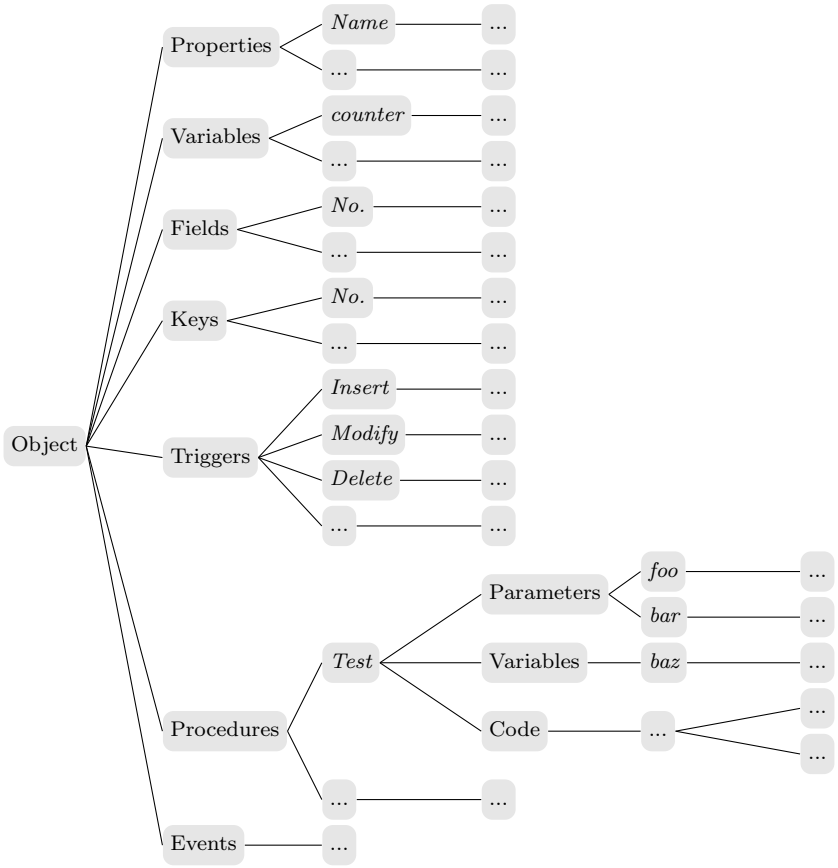
Given a code base, Supervisionproduces an interactive and hyperlinked set of HTML pages with the following main features:

- Hyperlinks
  - Usages of user-defined functions, variables, parameters, fields, system-defined variables, and object types link to their declaration
  - Usages of C/AL system functions link to official Microsoft C/AL language documentation
- Tool-tips
  - When hovering over a declaration, usages within the same object is counted and usages are high-lighted
  - When hovering over usages of variables, parameters etc, the type of the corresponding declaration is displayed
- Indexing
  - An object index page provides overview of the NAV objects processed
  - For individual C/AL objects, the object is outlined, enabling easy navigation to e.g. user-defined functions
- Code colouring: Detailed and customisable code colouring
- View options
  - Hide/show comments
  - Enable/disable tool-tips
- Navigation history is listed in-page for easy back and forth navigation

3 Structure of C/AL object files

The structure of C/AL object files are similar but vary from object type to object type. A table object has fields and keys, whereas a codeunit object does not. Likewise, a page object has controls whereas neither a table nor a codeunit has such a construct.

To get a sense for how a typical C/AL object file may look, consider the simplified and partial Abstract Syntax Tree (AST) depicted below:



The depicted AST is not full-fidelity as it would not fit on a page.

4 Architechture

The application is comprised of a set of components that compile to separate DLLs. The following UML Component Diagram shows the main dependencies between these DLLs.

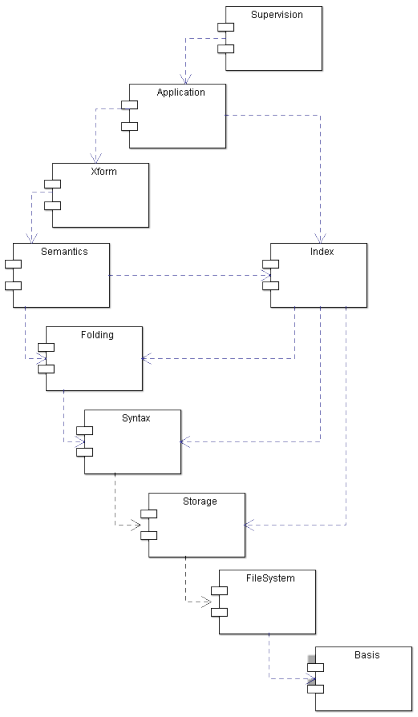


Figure 1: Components with main dependencies

The following sections will discuss the main components with focus on F#.

4.1 Basis

The basis module contains general-purpose types and functions for use in main components of Supervision.

Quite early in the development it became clear that we needed to handle processes gracefully. A process can be copying a file, indexing a source code container, doing semantic analysis, etc. Common to all these processes are that they will typically involve noticable latency. Several questions arise:

- How do we handle latency?
- How do we handle cancellation?
- How do we handle exceptions?
- How do we handle progress updates?

The Reactive Extensions for .NET (R_X) library [6] provides a solution to some of these problems.

The generic `IObservable` type is the dual of the `IEnumerable` type. Whereas `IEnumerable` models things that can be enumerated, `IObservable` models things that can be observed. The difference is pull vs. push. `IEnumerable` is a pull-based collection type whereas `IObservable` is push-based: instead of having to pull out elements one by one, the elements are "pushed" to the subcriber, when they have been computed.

Both types can solve the latency issue to some degree. `IEnumerable` can provide an `IEnumerator` and this allows the client to take one element at a time, but the client can stop at any time without taking the rest of the elements. `IObservable` is better, however, because the subscriber does not even have to wait for the first element to arrive, it can do other things until the element is computed. This is the concurrent nature of R_X. Observing elements can be done on a thread other than the generating thread. R_X provides combinators such as `SubscribeOn` and `ObserveOn` to help with this very problem.

4.1.1 Push/Pull

The next few paragraphs we will briefly show a little closer how the push-pull styles relate. This discovery was made by Erik Meijer and a more detailed explanation is given in an interview [7] on Microsoft's Channel 9, where Brian Beckman interviews Erik Meijer on the origin and design of R_X.

4.1.2 Pull

The full interface definitions of the `IEnumerable`/`IEnumerator` types are as follows:

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current { get; }
}
```

In pseudo-code, the `IEnumerable`/`IEnumerator` types may be defined as:

$$\overbrace{() \longrightarrow \underbrace{(() \longrightarrow a)}_{enumerator}}^{enumerable}$$

In other words, it is a function which takes nothing as input and produces another function which takes nothing as input but then produces an output value of some type. The second function can be called repeatedly until there are no more elements or an exception is thrown.

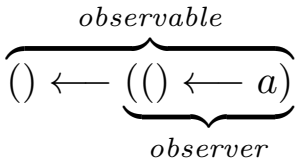
4.1.3 Push

The full interface definitions of the `IObservable`/`IObserver` types are as follows:

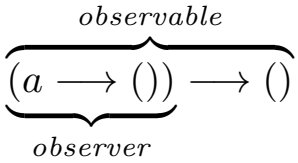
```
public interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

public interface IObserver<in T>
{
    void OnCompleted();
    void OnError(Exception error);
    void OnNext(T value);
}
```

In pseudo-code, the `IObservable`/`IObserver` types may be defined as the dual of `IEnumerable`/`IEnumerator`.

First, we reverse the arrows:



Then we normalize the notation to use left-to-right style:



Cancellation is also supported by `IObservable`. Subscribing to an `IObservable` returns an `IDisposable`. If the `IDisposable` is disposed the subscription is terminated.

4.1.4 Processes

Now that we have discussed the push-pull-styles, we will show how the push-style can be further nuanced in a `Process` abstraction.

In fact we model a process in terms of R_X:

```
type 'a Process = 'a Signal IObservable
```

where a signal is defined as:

```
type 'a Signal =
    | ValueSignal of 'a
    | ErrorSignal of Exception
    | StatusSignal of string
    | ProgressSignal of decimal
```

The interesting idea here is that if we slice a process into a stream of events, the client can subscribe to the process, pattern match over signals and listen to the values of interest.

Let's suppose a process processes a sequence of files and that for some reason one file cannot be accessed, thus yielding a file system exception. In this case it is often better to continue, while informing the user, than to stop the entire process. The user can then decide whether to stop and investigate or continue processing. In this case the process would stream in an error-encapsulating signal which the observer can choose to ignore or break on.

An added benefit of our process definition is that it provides a uniform interface which can be applied to a GUI or console application for any kind of process, regardless of what type of element it produces—even unit. Message and progress values are directly applicable to update progress bars and status labels.

It is also interesting to ponder that processes can be "multi-resolutional" but currently our abstraction does not directly model this and it seemed over-kill for our application. Several interesting applications make use of multi-level processes and this could be an interesting type to have.

4.2 FileSystem

The FileSystem module provides a virtual file system abstraction. At present there are only a few concrete implementations but the abstraction is sufficiently expressive to model many kinds of file systems. The first implementation is the`NativeFileSystem`. This implementation uses .NET methods to access the underlying file system.

A virtual file system abstraction should take into account variable latencies as well as read-only file systems. The current abstraction does not provide a complete solution to this problem but the path is fairly clear: potentially high-latency operations are exposed as processes via the process abstraction described in the basis section.

In the future we also envison using nested file systems. One example could be a zip file on an FTP server. This is, however, not core to Supervisionand does not have priority at present.

4.3 Storage

The storage module provides abstractions for transparent access to source-code objects in some form of container. The core abstraction is simple, but sufficiently general to meet our needs. The core abstraction is the `ObjectStore` type. Key features of this type is access to source code and enumeration of source code objects.

The `FileSystemObjectStore` implementation class provides access to objects laid out in virtual file system in a conventional pattern.

4.4 Syntax

Mastering the syntax of a language is a prerequisite for doing any kind of serious static analysis. Our biggest challenge in that regard was that the C/AL

language does not have a formally defined grammar. The most precise description available is the on-line ref that Microsoft provides on MSDN [5]. We therefore had to resort to a trial and error approach and rely on testing the parser on large sets of C/AL objects for verification. At the time of writing, there seems to be only a few minor issues left.

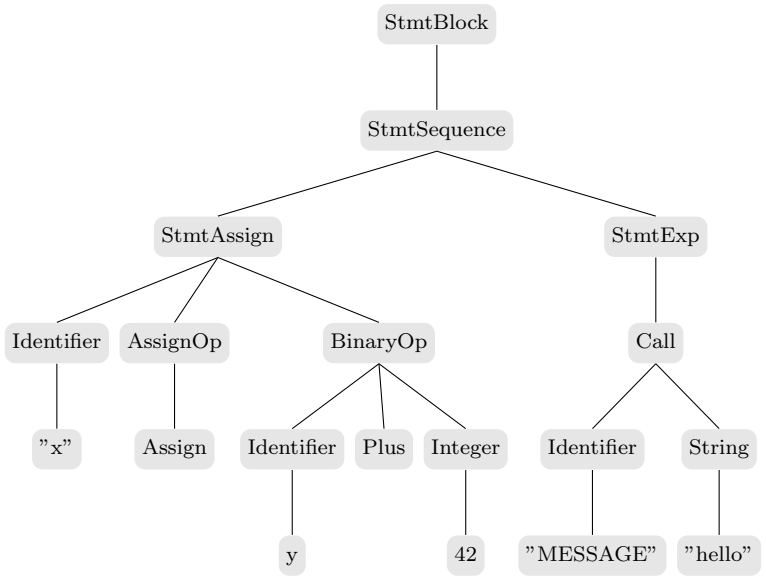
For the handling of syntax, we have taken a very traditional approach, building Abstract Syntax Trees (AST) using F# PowerPack's [8] fslex and fsyacc for lexing and parsing, respectively. In case you are not familiar with parsing technologies, it suffices to know that lexing produces a sequence of tokens from the source file, which is then used by parser production rules to produce AST nodes.

4.4.1 Abstract Syntax Tree

To get a feel for how the ASTs are structured, let's take a look at a simple code fragment:

```
BEGIN
  x := y + 42;
  MESSAGE('hello')
END
```

For this code fragment, the parser produces the AST illustrated below. For simplicity, we have left out position information which will be discussed shortly.



A straigt-forward representation ASTs in F# is to use a combination of discriminated unions and tuples. As we also intent to also build a tool that can check code layout rules, we need source code position information to be available in the AST. The declarations below illustrate the approach taken:

```
type Position =
    {
        Token : string
        Line : int
        Column : int
        StartIndex : int
        StopIndex : int
    }

type Positions = Position list

type Statement =
    | StmtEmpty
    | StmtSequence of Statement * Statement * Positions
    | StmtBlock of Statement * Positions
    | StmtAssign of Expression * AssignOperator * Expression *
        Positions
  ...
```

Position lists have an entry per terminal symbol identified in the source code. Thus, for statement sequences like `S1; S2`, which corresponds to the discriminator StmtSequence, there is a single entry for the semicolon separating the two statements. Similarly, statement blocks, which consist of statements between keywords `BEGIN` and `END`, have two position entries corresponding to `BEGIN` and `END`.

4.4.2 Lexing

The structure of C/AL source code text files is fragments of C/AL statements and expressions packaged into a container that holds meta-data about the objects themselves. This—combined with the fact that reserved words in C/AL are not really reserved—makes lexing a bit tricky. One example of this is the reserved word "Permissions". This word is marked as reserved in the documentation because it is used in the meta-data to define permissions for the object. However, variables can actually be named "Permission", so this poses the problem that at lexing time, we need the string "Permission" to produce two different tokens—either the keyword or an identifier—depending on whether we are inside the container or inside a code section (statement or expression). To work around this issue, we use fslex's `BufferLocalStore` to keep trace of wheather or not we are in a code section.

4.4.3 Parsing

Parsing C/AL with fsyacc is pretty much "by-the-book". The example below shows the `Statements` rule. This rule states that `Statements` can be matched by a single statement—returning the single statement matched (represented by $1)—or a `Statement` followed by a semicolon, followed by `Statements`, in which case we return a `StmtSequence` discriminator consisting of the two matched statements (represented by $1 and $3) and the position of the semicolon.

```
Statements :
    | Statement { $1 }
    | Statement T_SEMICOLON Statements
      { StmtSequence (
            $1,
            $3,
            TokenPositions([ (2, "T_SEMICOLON") ], parseState)) }
```

Apart from the C/AL language not having a formal grammar, the biggest challenge was to get decent error messages. To handle lexing and parsing exceptions, fsyacc lets you define a `parse_error_rich` function, which is called when an error occurs. However, the function does not return anything and throwing an exception from the function does not work either. Therefore, we found it necessary to introduce a mutable variable to hold the error message, as illustrated in the code below, where we have defined the inline "`++`" for `StringBuilder`'s `append`.

```
let mutable ErrorContextDescriptor : string = ""

let parse_error_rich =
    Some (fun (ctxt: ParseErrorContext<_>) ->
        let sb = new StringBuilder()
        let nl = Environment.NewLine
        sb
        ++ (sprintf "CurrentToken: %A" ctxt.CurrentToken) ++ nl
        ++ (sprintf "Message: %s" ctxt.Message) ++ nl
        ++ (sprintf "ReduceTokens: %A" ctxt.ReduceTokens) ++ nl
        ++ (sprintf "ReducibleProductions: %A"
                    ctxt.ReducibleProductions) ++ nl
        ++ (sprintf "ShiftTokens: %A" ctxt.ShiftTokens) ++ nl
        ++ (sprintf "StateStack: %A" ctxt.StateStack)
        |> ignore
        ErrorContextDescriptor <- sb.ToString()
    )
```

Notice that the error message produced by the above code includes detailed context information such as `ShiftTokens` and `StateStack`.This information is valuable when debugging the parser, as it can be combined with fsyacc's listing function to provide visibility into why exactly the parser failed to parse the in-

put. The listing function of fsyacc is enabled with the "`-v`" command line switch, which will make fsyacc dump a textual representation of the non-deterministic finite automaton that it creates.

4.5 Folding

In building the Supervisionapplication, it has been sufficient for us to base our static analysis directly on the ASTs produced by the parsing process. However, we quickly found that coding directly against the AST resulted in a lot of similar code, and it was clear that we needed a functional equivalent of the visitor patterns known from object-oriented languages [9]. A bit of research taught us that the answer was *folding*, or *catamorphisms*.

In its simplest form, folding is known from list functions built into most functional languages. The idea is pretty simple. To fold a list, you need an initial value and a function that can combine the result so far with the next element of the list. The typical text book example below calculates the sum of all elements of the list.

```
let xs = [1; 2; 3]
let sum = List.fold (fun x y -> x + y) 0 xs
```

Folding can be generalised from working on lists to working on trees, as described in five excellent blog entries by Brian McNamara [10, 11, 12, 13, 14]. As ASTs are exactly trees, folders can also be defined for ASTs. One challenge in doing so is that a combiner function is needed for each node type of the tree, and for our AST that means approximately 140 functions! To manage that many functions, we grouped the functions into natural groups and defined a record type for each group.

However, even with the functions grouped, each function still needs to be defined before using the folding can begin. Our solution to that challenge was to introduce a `SuperCombiner`, which takes an initial value and a combiner function and implements "default" values for all functions: initial values for leaves in the tree, and combined results for non-leaves. The context parameter, `ctx`, will be explained later.

```
let SuperCombiner
        (ctx : 'c)
        (initial : 'r)
        (combine : 'r -> 'r -> 'r) : Combiners<'c,'r> =

    let inline (++) a b = combine a b

    let StmtCombiners : StmtCombiners<'c,'r> =
        {
            fStmtSequence = fun ctx s1acc s2acc p ->
```

```
                                    s1acc.Result ++ s2acc.Result
        fStmtEmpty = fun ctx -> initial
        ...
    }

...

let super : Combiners<'c,'r> =
    {
        StmtCombiners = StmtCombiners
        ...
    }

super
```

Given the `SuperCombiner`, various combiners can easily be defined. Below, you will find implementation of `CountCombiners` and `SeqCcombiners` which can be used as "templates" in situations where we want to count occurrences of "something" or collect results in a sequence over the whole AST.

```
let CountCombiners =
    SuperCombiner () 0 (fun a b -> a + b)

let SeqCombiners<'c,'r> =
    SuperCombiner
        ()
        Seq.empty<'r>
        (fun a b -> Seq.append a b)
```

Below is a simple example of a rather silly analysis, which simply counts the number of semicolon statement separators. Notice that by using the `CountCombiners` as a template, we only need to "override" the functions that differ from the default implementation provided by `CountCombiners`.

```
let combiners =
    { CountCombiners with
        StmtCombiners =
            { CountCombiners.StmtCombiners with
                fStmtSequence =
                    fun ctx s1acc s2acc p ->
                        1 + s1acc.Result + s2acc.Result;
            }
    }
```

Catamorphisms work buttom-up, producing results at the lower levels and combining on the way up towards the root. This proved insufficent for doing semantic analysis, the reason being that e.g. variables can be defined at various levels—for example local to a C/AL function or globally for the object. We

therefore needed to extend the folder to be able to build a context while calling recursively down the AST. Plesae refer to section 4.7 for more details.

Even though the folder are performing well, we found it wasteful to traverse the whole AST, in cases where we are only interested in certain parts of the AST. We have therefore built recursion control into the folder, such that recursion can stop at any level, potentially depending on the context. Two recursion controls are especially useful: full recursion and "only to C/AL language level", i.e. everything except expressions and statements.

```
let FullyRecursive : RecursionControl =
    {
        recurseExpression = fun _ -> true
        recurseStatement = fun _ -> true
        ...
    }

let NoCal : RecursionControl =
    { FullyRecursive with
        recurseExpression = fun _ -> false
        recurseStatement = fun _ -> false
    }
```

4.6 Indexing

Indexing is the process of listing all constructs of an object that can be referred internally as well as externally. We index the full object store to optimize finding declarations and references across objects. The primary client of the index is semantic analysis—see section 4.7.

As an example, consider two codeunits, where one calls a procedure in the other:

```
OBJECT Codeunit 50000 Calculator
{
  OBJECT-PROPERTIES
  {
    Date=31-10-12;
    Time=17:16:09;
    Modified=Yes;
    Version List=;
  }
  PROPERTIES
  {
    OnRun=BEGIN
          END;

  }
```

```
  CODE
  {
    VAR
      globalVar@1112800000 : Integer;

    PROCEDURE Calculate@1112800000(param@1112800000 :
       Integer) returnVar : Integer;
    BEGIN
      returnVar := globalVar + param + 42;
      EXIT
    END;

    BEGIN
    END.
  }
}

OBJECT Codeunit 50001 Process
{
  OBJECT-PROPERTIES
  {
    Date=31-10-12;
    Time=17:21:35;
    Modified=Yes;
    Version List=;
  }
  PROPERTIES
  {
    OnRun=BEGIN
            Process()
          END;

  }
  CODE
  {
    PROCEDURE Process@1112800001();
    VAR
      calculator@1112800000 : Codeunit 50000;
    BEGIN
      MESSAGE('Value is: ' + FORMAT(calculator.Calculate
          (117)))
    END;

    BEGIN
    END.
  }
}
```

When resolving `Calculate` in `calculator.Calculate()`, we use the index to search for the name "`Calculate`" inside `Codeunit 50000`.

The following constructs are indexed:

- Properties
- Triggers
- Fields
- Keys
- Procedures
- Events
- Controls

Another usage of indexing is for finding usages of e.g. Procedures. For example, if we want to find all usages of `(Codeunit 5000).Bar`, then a natural first step is to find all objects that reference `Codeunit 50000` at all.

The indexer supports both inbound and outbound links, although the Supervisionapplication only uses outbound links, correspoding to forward linking. The example below shows the succinct expressiveness of the folder. Given the AST representation of a NAV object declaration, source, the function below returns a sequence of object references to source. In C/AL, an object is referenced through properties, variables, or parameters.

```
member private this.IndexUsages (source : NavObject)
              : ObjectReference seq =
    let combiners =
        {
            SeqCombiners<unit, ObjectReference> with

                PropertyCombiners =
                    {
                        SeqCombiners.PropertyCombiners with
                            fObjectReferenceProperty =
                                fun ctx name oref p ->
                                    seq { yield oref }
                    }

                CodeCombiners =
                    {
                        SeqCombiners.CodeCombiners with
                            fVariableDeclaration = fun ctx id t p ->
                                match t with
```

```
                        | ObjectReferenceType oref ->
                            seq { yield oref }
                        | _ ->
                            Seq.empty

                    fParameterDeclaration = fun ctx pe id t p
                        ->
                        match t with
                        | ObjectReferenceType oref ->
                            seq { yield oref }
                        | _ ->
                            Seq.empty
                }
        }

    source.Fold
        ()
        IdentityContextBuilder
        RecursionControl.NoCal
        Seq.empty
        combiners
```

Notice also that we call folder with `RecursionControl.NoCal` to avoid folding over expressions and statements.

4.7 Semantics

Correct name resolution is a prerequsite for functionality such as "go to declaration" and "find usages" as well as rules like "unused variable" or "unused procedure". As C/AL does not specify name resolution rules, we again had to resort to trial and error in figuring out the name resolution rules. At the time of writing, the algorithm is not 100% complete, but we believe that what we resolve is correct. At least we have found several examples where our algorithm is more accurate than "Developer's Toolkit"—a tool provided by Microsoft, which has recently been discontinued.

Names that need resolving in C/AL are names of objects, variables (local and global), procedures, parameters, return variables, table field names, system functions, system function name spaces, and controls. We use the following type for the result of resolving a single name:

```
type DeclarationReference =
    | VariableDeclarationReference of TypedIdentifier
    | ParameterDeclarationReference of TypedIdentifier
    | ReturnDeclarationReference of TypedIdentifier
    | ProcedureDeclarationReference of ProcedureHeader
    | FieldDeclarationReference of TypedIdentifier
    | SystemFunctionDeclarationReference of SystemFunction
```

```
    | SystemFunctionNamespaceDeclarationReference of
        SystemFunctionNamespace
    | ExternalReference of ObjectReference * DeclarationReference
    | ObjectReference of ObjectReference
    | ControlDeclarationReference of ControlIndexNode
    | UnresolvableReference of Identifier * string
```

C/AL supports shadowing of names, so two usages of the same name can resolve to different declarations. We therefore need to keep track of the some sort of context as the folder recurses through the AST. Specifically, we are interested in keepting trace of AST nodes where identifiers can be declared. An example of an AST node that is relevant for name resolution is ProcedureDeclaration, because this is where local variables, parameters, and (optionally) named return variable are declared. The type `NodePath` (defined below) is essentially a path from the top of the AST down through nodes that are relevant for name resolution.

```
type Step<'a> =
    {
        Node : 'a
        Up : NodePath
    }

and DotExpression =
    {
        Lhs : Expression
        Rhs : Identifier
    }

and NodePath =
    | Top
    | ObjectStep of Step<NavObject>
    | ProcedureStep of Step<ProcedureDeclaration>
    | DotStep of Step<DotExpression>
    | WithDoStep of Step<Expression>
    | TriggerStep of Step<TriggerDeclaration>
```

Overall, our name resolution algorithm searches backwards up through the `NodePath` until the name is found. At each step we search the relevant declarations in the correct order. The code below illustrates how we search for names declared at procedure level. The essence is that the search order is local variables, then parameters, and finally named returns.

```
...
| ProcedureDeclaration (
    (_, _, parameterDeclaration, returnType, _),
    (variableDeclarations, _, _), _) ->
    this.Search usage (
```

```
seq {
    yield VariableDeclarations.Search variableDeclarations
    yield parameterDeclaration.Search
    yield ReturnType.Search returnType
})
```

4.8 Xform

This component transforms an object store of C/AL objects into a set of HTML pages. The process performs the following steps:

- Build index
- Create JSON index
- Create HTML Pages (one per C/AL object)
  - Load object from object store
  - Parse object
  - Create source code annotations using folding and semantic analysis
  - Apply source code annotations producing HTML markup
  - Save annotated source code as HTML file in virtual file system
- Create HTML index page

4.9 Application

This component implements application scenarios. An application scenario is defined as a `Process`. The primary process of Supervisionis defined as:

```
SourceToHtmlScenario
    (store : ObjectStore, assets : FileSystem, output : FileSystem)
    : ObjectReference Process
```

There are several application scenarios, but Supervisiononly exposes the source to HTML scenario.

The scenario function signature abstracts the physical location of the object source files and assets (JavaScript and CSS documents) as well as the target storage file system. The virtual file system is responsible for abstracting these implementation details.

Since application scenarios return processes, they can be applied directly to both console and GUI applications, where the application context can be completely oblivious to the underlying implementation, that is to say, application scenarios are interchangeable due to the unified process interface.

4.10 Supervision

The SupervisionGUI was created using C# and the Visual Studio RAD designer. This was necessary because the Visual Studio F# environment does not have a RAD designer. Subsequently, the code was then translated to F# by hand in order to have the whole application written in one language.

5 Future Developments

The components are sufficiently generalised to enable rapid development of new static analysis features. The most challenging development aspect is the GUI. Below, some ideas for future development:

- Find usages: Supervisionprovides find usages inside an object, and forward-linking is across objects, but there is no find usages functionality across objects as known from Microsoft Dynamics NAV Developer's Toolkit, which is no longer supported
- Checking of coding conventions: Formatting conventions of Microsoft Dynamics NAV C/AL Programming Guide
- Checking of potential run-time errors like "string overflow"
- Reformatting of code so it conforms to Microsoft standards
- Refactoring facilities like "introduce local function"

6 F# Experiences

In general, we have enyoued the succint nature of the F# programming language and its emphasis on immutable and side-effect free computation, while still enabling effortless I/O using traditional imperative constructs.

However, the Visual Studio F# development environment appears rudimentary compared to the C# development environment. In particular, the lack of "find usages", refactorings, and a RAD designer, have been painful and slowed our development velocity.

It was noticable, however, that the harder-to-debug problems we encountered were often due to the interleaving of side-effecting computations with otherwise pure functional code. In particular, we have had some challenges with R_X—specifically exception handling and working with computation expressions; for example, having the need for two interpretations of a for loop (push and pull style within an R_X computation expression.)

In general, the performance is very good, but along the way, we have had several challenges. As a trivial example, we discovered that `printf` with "`%A`" significantly impacted performance, even when used for printing values of very simple discriminated unions. Pin-pointing this took us quite a while as the Visual Studio performance profiler leaves much desired.

Working with fslex and fsyacc was relatively easy once we found out how to debug with the "`-v`" option and getting the state printed in case of exceptions. Our target platform is .NET 4.0, so we had to recompile F# PowerPack to make it work.

During the project we developed a considerable amount of unit tests (116 at the time of writing.) Our first choice for unit test framework was MsTest, but we did not succeed in getting it up running, so we ended up using xUnit.net [15], which worked out of the box and integrates well with Visual Studio.

We found the performance profiler of Visual Studio to be insufficient in pin-pointing performance issues. Instead, we successfully trialled dotTrace [16] by JetBrains.

7 Conclusion

By building Supervision, we have shown that F# is indeed very well-suited for static analysis. Syntax is elegantly expressed using discrimanated unions, and higher-order functional patterns like folders/catamorphisms are extremely powerful and general. The combination of folders and record type templates proved especially suitable to scale static analysis to real-life sized languages. Looking back, it also strikes us that the vast majority of code written is general-purpose for static analysis of C/AL.

Being part of the .NET family of langages, F# enables reusing libraries and components written for .NET in other languages. In particular this is useful when looking for efficient (mutable) data structures for performance critical parts of an application.

We have experienced F# to be error-free, but Visual Studio lacks features and maturing to lift F# to the standards of C#.

8 References

[1] Checkstyle 5.6.
http://checkstyle.sourceforge.net/,
2012.

[2] FindBugs 2.0.1—Find Bugs in Java Programs.
http://findbugs.sourceforge.net/,
2012.

[3] StyleCop 4.7.
http://stylecop.codeplex.com/,
2012.

[4] Microsoft Dynamics NAV.
http://www.microsoft.com/en-us/dynamics/erp-nav-overview.aspx,
2012.

[5] C/SIDE Reference Guide.
http://msdn.microsoft.com/en-us/library/dd301468.aspx,
2009.

[6] Reactive Extensions.
http://msdn.microsoft.com/en-us/library/hh242985(v=vs.103).aspx.

[7] Microsoft Channel 9 Expert to Expert: Inside the .NET Reactive Framework (R_X).
http://channel9.msdn.com/Shows/Going+Deep/Expert-to-Expert-Brian-Beckman-and-Erik-Meijer-Inside-the-NET-Reactive-Framework-Rx,
2009.

[8] F# PowerPack.
http://fsharppowerpack.codeplex.com/,
2012.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*.
Addison-Wesley Professional, 1 edition, November.

[10] Brian McNamara. Catamorphisms, part one.
http://lorgonblog.wordpress.com/2008/04/05/catamorphisms-part-one/,
2008.

[11] Brian McNamara. Catamorphisms, part two.
http://lorgonblog.wordpress.com/2008/04/06/catamorphisms-part-two/,
2008.

[12] Brian McNamara. Catamorphisms, part three.
http://lorgonblog.wordpress.com/2008/04/09/catamorphisms-part-three/,
2008.

[13] Brian McNamara. Catamorphisms, part four.
http://lorgonblog.wordpress.com/2008/05/24/catamorphisms-part-four/,
2008.

[14] Brian McNamara. Catamorphisms, part five.
http://lorgonblog.wordpress.com/2008/05/31/catamorphisms-part-five/,
2008.

[15] xUnit.net.
http://xunit.codeplex.com/,
2012.

[16] JetBrains. dotTrace 5.2.
http://www.jetbrains.com/profiler/.