TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

IAPM02/18
Ian Erik Varatalu 203952IAPM

F# Type Provider and Compiler for the AL Programming Language

Masters’s Thesis

Supervisor: Juhan-Peep Ernits
PhD

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

IAPM02/18

Ian Erik Varatalu 203952IAPM

F# tüübitekitaja ja kompilaator AL programmeerimiskeelele

Magistritöö

Juhendaja: Juhan-Peep Ernits

PhD

Tallinn 2022

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, the literature and the work of others have been referenced. This thesis has not been presented for examination anywhere else.

Author: Ian Erik Varatalu

08.05.2022

Abstract

The AL language is a Pascal-based programming language used in the Enterprise Resource Planning (ERP) system Microsoft Dynamics 365 Business Central, that was shaped during the development of Navision 3.x in the 1990s and has since been further developed under the constraint of maintaining backwards compatibility.

The language is good for database queries and business logic, as it is easy to learn and master, and remains relevant with its large network of partners, who customize the platform further to fit the needs of customers. On the other hand, because of its long heritage and the requirement of remaining backwards compatible, the language has many limitations and missing safety-checks compared to modern programming languages.

In the current thesis we analyze the limitations of the AL programming language by defining 13 issues, and set out to address the limitations by embedding the development into a modern .NET language – F#. The goal of the embedding is to provide developer convenience and type safety features of F#, while still remaining compatible with the large existing codebase. The issues have been aggregated into 8 requirements which constitute the specification of the proposed solution.

Within the thesis, we develop a compiler for transforming the F# programming language to AL. Using the compiler we implement modern features that were previously not possible in the AL language, such as type inference, inheritance and deserialization, and guarantee type safety in places where it previously was not done. Then we compare the created alternative development workflow to the traditional AL development workflow.

In order to utilize the large existing AL code base we additionally develop an AL type provider for convenient integration of existing code with the developed application.

The outcome of the work is demonstrated and evaluated in 2 end-to-end use cases, one involving the development of HTTP requests involving type provider assisted type safe processing of structured data in JSON format and the other one involving designing tables with shared fields and procedures by making use of Object-oriented design of Record types.

The thesis is written in English and is 74 pages long, including 6 chapters, 7 figures, 30 code listings, and 3 tables.

Annotatsioon

AL on Pascalil põhinev programmeerimiskeel, mida kasutatakse ettevõtte ressursside planeerimise (ERP) süsteemis Microsoft Dynamics 365 Business Central. Keel kujunes 1990. aastatel Navision 3.x arendamise käigus ning seda on sellest ajast täiendatud pidades oluliseks tagasiühilduvust.

AL põhitugevus on kasutajate sihtgrupile päringute ja äriloogika kirjeldamiseks, kuna keel on lihtne ning seda kasutab suur partnerite võrgustik, kes kohandavad AL keeles rakendusi klientide vajadustele vastavaks. Teisest küljest on sellel keelel tagasiühilduvuse nõude tõttu palju puudujääke, mida oleks tänapäevastes programmeerimiskeeltes lihtne saavutada.

Käesolevas töös analüüsime AL-i programmeerimiskeele piiranguid defineerides 13 probleemi ja loome võimaluse arendada AL rakendusi moodsa .Neti põhise programmeerimiskeele F# keskkonnas. F# kasutuse eesmärk on pakkuda arendajale mugavust ning automaatseid turvalisusfunktsionaalsusi, jäädes samas ühilduvaks suure olemasoleva AL koodibaasiga. Probleemid on koondatud 8-ks nõudeks, mis moodustavad pakutava lahenduse.

Lõputöö raames töötame välja kompilaatori F# programmeerimiskeele teisendamiseks AL-i. Kasutades kompilaatorit rakendame tänapäevaseid keele funktsioone, mis polnud varem võimalikud, nagu tüübi järeldamine, pärimine ja deserialiseerimine, ning tagame tüübiohutust kohtades, kus seda varem ei tehtud. Seejärel võrdleme loodud alternatiivset arendustöövoogu traditsioonilise AL-i töövooga.

Suure olemasoleva AL koodibaasi kasutamiseks arendame lisaks välja AL tüübitekitaja olemasoleva koodi mugavaks integreerimiseks arendatud rakendusega.

Töö tulemust demonstreeritakse ja hinnatakse kahel algusest-lõpuni kasutusnäitel. Esimene näide hõlmab HTTP päringute väljatöötamist, kasutades tüübitekitaja abiga struktureeritud andmete töötlemist JSON formaadis. Teine näide hõlmab jagatud väljade ja protseduuridega tabeli kujundamist, kasutades objektorienteeritud disaini.

Lõputöö on kirjutatud inglise keeles keeles ning sisaldab teksti 74 leheküljel, 6 peatükki,

7 joonist, 30 koodinäidist, 3 tabelit.

List of abbreviations and terms

.NET	Open source developer platform by Microsoft
AL	Application Language - the language used to develop applications for Business Central
API	Application Programming Interface
Business Central	An ERP business software previously known as Dynamics NAV
C/AL	Client Server Application Language - AL's predecessor used in 2018 and before
ERP	Enterprise Resource Planning
F#	Modern .NET programming language used as the primary development language in the current work
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
XML	Extensible Markup Language

Contents

List of Figures

Listings

List of Tables

1 Introduction

AL is a database specific programming language based on Pascal and is primarily used for retrieving, inserting and modifying records in a Microsoft Dynamics 365 Business Central application database.

The language is only used for building Enterprise Resource Planning (ERP) applications for the Business Central platform, but such applications are very popular, and currently in use in close to 200 000 companies worldwide.

The language is good for specifying database operations and business logic, and having such a language for easy customization is one of the main reasons for its success. On the other hand, because of its long history and Pascal-based syntax, the AL language and type system has many limitations, lacks many safety-checks that could easily be achieved, and is very verbose for complex solutions: a scenario it is often used in.

The official solution to these limitations is to call cloud functions over the HTTP protocol. However, calling and reading the results of these external web services is also made difficult by the problems stated above.

The goal of the current thesis is to provide an alternative to develop AL code in F# in a way that is versioned, can integrate with the existing AL codebase, guarantee safety from more types of bugs than the AL compiler, and most importantly, is more concise and hopefully also clearer.

The current thesis is an elaboration of the ideas "Translation of C/AL to an object-oriented programming language" and "Extend C/AL with polymorphism/type inference or explicit subtyping" (Hvitved, 2009).

The source language, F#, is chosen for two reasons: First, the F# language lives in the same .NET ecosystem as AL, and second, F# is a common language to write compilers with, previously used for projects such as Fable[1], the F# to JavaScript compiler. Fable is also an inspiration for this project.

The steps for achieving the goal are as follows:

[1]Fable (2021). *Fable - JavaScript you can be proud of!* Accessed: 2021-11-23. URL: https://fable.io/.

- Describe typical issues an AL developer encounters during development
- Describe how these issues could be mitigated
- Establish requirements for an alternative solution
- Implement a way to use existing AL code in F#
- Implement a compiler turning F# into working AL Code

The approach is not intended to replace the entire AL language, but rather mitigate aspects of development that are very time-consuming and prone to errors in AL, and can later be integrated directly into the AL codebase.

2 The Problem

In the current chapter we discuss the background of AL, and present *issues*, to which we refer to as an important topic for debate or discussion in the AL Language.

We will largely rely on the paper Architectural Analysis of Microsoft Dynamics NAV (Hvitved, 2009) for technical explanations, as it is currently the only report about Microsoft Dynamics NAV (as Business Central was previously called) known by the author that is published in the computer science literature.

Some other important details to keep in mind:

- No formal specification of the AL language is public, but there is a formalization of an older version called C/AL in (Hvitved, 2009).
- Almost all functionality of Business Central is currently written in AL (over 2 million lines of source code in AL).
- AL (and C/AL) was not originally targeted at computer scientists / software engineers, and most AL developers do not have a background in software development.
- The application has to maintain backwards compatibility, and Microsoft has to be very careful to not break any existing code - so it's important to remember that many of these upcoming issues exist purely for preserving legacy code.

The book Programming Microsoft Dynamics 365 Business Central (Brummel, M. et al., 2019) has a good overview of the initial design philosophy behind C/AL and AL. As stated by the developer of the original AL compiler, runtime and IDE, Michael Nielsen, in (Brummel, M. et al., 2019), the goals of the language design included:

- Allowing the developer to focus on design rather than coding, but still allowing flexibility
- Providing a syntax based on Pascal stripped of complexities, especially relating to memory management
- Providing a limited set of predefined object types and reducing the complexity and learning curve

- Implementing database versioning for a consistent and reliable view of the database
- Making the developer and end user more at home by borrowing a large number of concepts from Office, Windows, Access, and other Microsoft products

As C/AL started out very simple, many features have been added to the language as an afterthought - and many features have been deprecated. Because of these gradual changes, there are some design quirks in the system.

As another side effect of the long history of developing the system in a certain way, the application has an unnormalized database design. According to (Hvitved, 2009), the tables in the application were very large (the Item table had 175 columns) and sparse (many null values). Bear in mind that these 175 columns were there in 2009 before any custom development. Today, 13 years later, this number of columns in the Item table is down to 145, and still sparse.

Like these giant tables, some patterns are so deeply engrained into the ecosystem, that it is not worth forcing them into a new design. For these reasons, the upcoming solutions are oriented towards alleviating the issues in the current setting, rather than starting from scratch.

2.1 The Type System of AL

AL is a statically typed language, that is object-based (not to be confused with object-oriented), which means that there are no custom-defined types, but developers can create subtypes of Objects, such as Table, Page or Codeunit (which is an object type for business logic). There is no further subtyping, but there is a recently added Interface type, which we will describe later.

All types fall into one of two categories - Simple Types and Complex Types. Simple Types are data structures that can be serialized into a single database column. Complex Types are everything else. For a more complete description refer to Figure 1.

As the AL language is currently implemented in C# (previously in C++ (Brummel, M. et al., 2017)), many types are imported directly from the .NET ecosystem using thin wrappers, e.g. the type TextBuilder is a .NET StringBuilder under the hood. Since there have been some major changes from C/AL to AL, we marked the differences between the grammar of C/AL (Hvitved, 2009) and the current grammar of AL below.

The types listed in bold (e.g. **JsonValue**) have been added to the language since the change from C/AL to AL. Types with struck out text (e.g. ~~Form~~) have been removed since the change from C/AL to AL. A full reference of types with descriptions in AL can be found in the official documentation[1].

[1]Microsoft Docs (2022a). *Data Types and Methods in AL*. en. Accessed: 2022-4-6. URL: https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/library.

⟨*Type*⟩	::=	⟨*SimpleType*⟩	
	|	⟨*ComplexType*⟩	

⟨*SimpleType*⟩	::=	Integer	
	|	Text[*n*]	String of size *n*
	|	Code[*n*]	String of size *n* - case-insensitive
	|	Char	
	|	Decimal	
	|	Option	
	|	Boolean	
	|	Date	
	|	Time	
	|	DateTime	
	|	~~Binary[*n*]~~	Binary type - removed in AL
	|	Char	
	|	BLOB	Binary object of size *n*
	|	DateFormula	Multilanguage date calculation
	|	TableFilter	Type used for security filtering
	|	BigInteger	
	|	Duration	Time between two DateTime's
	|	GUID	Global Unique Identifier
	|	RecordID	
	|	*legacy enum types*	Built-in enumeration types
	|	**List of *SimpleType***	.NET List for simple types
	|	**Dictionary of *Key*, *Value***	.NET Dictionary for simple types
	|	**TextBuilder**	.NET StringBuilder wrapper
	|	**HttpHeaders**	.NET Dictionary wrapper
	|	**JsonValue**	
	|	**JsonObject**	
	|	**JsonArray**	
	|	**JsonToken**	JsonValue, -Object or -Array
	|	**XmlAttribute**	.NET equivalent wrapper
	|	**XmlAttributeCollection**	.NET equivalent wrapper
	|	**XmlText**	.NET equivalent wrapper
	|	**XmlNamespaceManager**	.NET equivalent wrapper
	|	**XmlDocument**	.NET equivalent wrapper
	|	**XmlElement**	.NET equivalent wrapper
	|	**XmlNode**	.NET equivalent wrapper
	|	**XmlNodeList**	.NET equivalent wrapper
	|	***other XML types***	.NET equivalent wrappers

⟨*ComplexType*⟩	::=	Action	
	|	RecordRef	Reference to *any* record
	|	Dialog	UI Dialog window
	|	Variant	Dynamic type
	|	InStream	
	|	OutStream	
	|	FieldRef	Reference to *any* field
	|	KeyRef	Reference to *any* key on a table
	|	File	
	|	⟨*ObjectType*⟩	See: *ObjectType* (below)
	|	array [*n1,...,nk*] of *Type*	Fixed-size array
	|	**ControlAddIn *n***	Client-side JavaScript plugin
	|	**HttpContent**	.NET equivalent wrapper
	|	**HttpRequestMessage**	.NET equivalent wrapper
	|	**HttpResponseMessage**	.NET equivalent wrapper
	|	**HttpClient**	.NET equivalent wrapper
	|	**Label *text***	Multilanguage text constant
	|	*DotNet *n*	.NET type (*removed in Cloud)
	|	***various other types***	Unknown
⟨*ObjectType*⟩	::=	Record *n* [temporary]	Cursor for a database type *n*
	|	Codeunit *n*	Business logic object of id *n*
	|	~~Form *n*~~	Legacy Windows client page *n*
	|	**Page *n***	Modern web-client page *n*
	|	Report *n*	Report printing definition *n*
	|	XmlPort *n*	Xml import or export definition *n*
	|	**Query *n***	Strongly-typed query definition *n*
	|	~~Automation *n*~~	Windows integration module type
	|	~~OCX *n*~~	Windows ActiveX module type
	|	~~MenuSuite *n*~~	Custom user-interface menu
	|	**Enum *n***	Enumeration type *n*
	|	PermissionSet *n*	Collection of user permissions
	|	Profile *n*	
	|	**EnumExtension *n***	Extension for Enum *n*
	|	**PageExtension *n***	Extension for Page *n*
	|	**TableExtension *n***	Extension for Record *n*
	|	**ReportExtension *n***	Extension for Table *n*
	|	**ProfileExtension *n***	Extension for Profile *n*
	|	**PermissionSetExtension *n***	Extension for PermissionSet *n*

Figure 1. BNF grammar of AL Types updated since Hvitved, 2009.

With the type system defined, we can now look into the first issue.

Issue 1. Collections, except fixed size arrays, can only contain simple types.

For a better understanding, consider the following collection variable declarations in the example below.

- ListOfIntegers - List of simple types, valid
- ListOfLists - List containing Lists of simple types, valid
- ListOfUris - List of complex types, not allowed
- MultidimensionalArrayOfUris - Fixed size array of complex types, valid

```
ListOfIntegers: List of [Integer];
ListOfLists: List of [List of [List of [Integer]]];
ListOfUris: List of [Codeunit Uri];
MultidimensionalArrayOfUris: array[500,500] of Codeunit Uri;
```

A special case of collections is the (Complex) Record type, which allows access to database table rows through a cursor-based iterator. Most commonly it is used like in the following example, first finding a set of records, and iterating over them inside a repeat loop.

```
if CustomerIterator.FindSet() then repeat
// function body
until CustomerIterator.Next() = 0;
```

The limitation means that there is no easy place to store instances of Complex types in code. For Codeunits non-fixed size storage is impossible. For Records there exists a special in-memory table structure where they can be re-inserted to.

2.2 Procedures and Variables

Methods, or procedures, as they're called in AL terminology, can be defined on the following types in Table 1 (below). The syntax of procedures has remained unchanged since (Hvitved, 2009).

There is another kind of procedure called trigger, that can be used for easy implementation of business logic, such as modification triggers - invoked every time a table is modified, or a validation trigger, invoked every time a user changes the value of a field. It is also possible to declare more triggers for extensibility of an application.

AL Type	Description
Table	Database table, referred to as Record in AL code
Page	Client-side page in the web application
Codeunit	Object type for encapsulating business logic
Report	Object type for printing documents, sometimes used as a user interface for procedures (i.e. "processing only reports")
XmlPort	Object type for importing and exporting large data structures via XML files
Query	Strongly typed database query that can be used as an Application Programming Interface (API) endpoint or a data source in code
Table Extension	Extension to a table in the application, used for customization
Page Extension	Extension to a page in the application, used for customization
Report Extension	Extension to a report in the application, used for customization
ControlAddIn	Client-side JavaScript plugin with AL interoperability

Table 1. AL Types that support procedures.

Issue 2. All local variables used in a procedure must be explicitly declared at the top.

Declarations of all local variables is something derived from the Pascal-based roots of the AL language, and while it is not a functional limitation, most modern programming languages do not require such declarations. One notable quirk about these variables is that the list often grows long, as many constructs in the language, such as for-loops, require explicit variable declarations for temporary storage.

```
local procedure CreateReservations(var OrderPromisingLine: Record ...
var
    ReqLine: Record "Requisition Line";
    SalesLine2: Record "Sales Line";
    ServLine2: Record "Service Line";
    JobPlanningLine2: Record "Job Planning Line";
    SalesLineReserve: Codeunit "Sales Line-Reserve";
    ServLineReserve: Codeunit "Service Line-Reserve";
    JobPlanningLineReserve: Codeunit "Job Planning Line-Reserve";
    ReservMgt: Codeunit "Reservation Management";
    SourceRecRef: RecordRef;
    ReservQty: Decimal;
    ReservQtyBase: Decimal;
    NeededQty: Decimal;
    NeededQtyBase: Decimal;
    FullAutoReservation: Boolean;
begin
...
```

Listing 1. Local variable declarations.

Issue 3. Procedures can only return a single variable

While a return value can be any variable, the amount of data that can be returned from a procedure is bound by the same limitations stated in Issue 1. It requires developers to pass mutable variables into functions for even simple tasks, and often requires the developer to declare the same variable multiple times - for every procedure it passes through.

For example, a common pattern in AL, as shown below, is to return a boolean value indicating whether the procedure succeeded, and storing results in reference parameters.

```
procedure ProcedureThatCanFail(var MutableValue:Integer) : Boolean
```

2.3 Dynamic Inputs

Issue 4. Dynamic parameter arrays

An issue adversely affecting the reliability of AL code is that AL makes it relatively easy to compile and publish code that does not work, as procedures often take dynamic parameter arrays for input.

For example consider one of the most used procedures in the entire AL codebase - "Get". Get is an instance method on the Record iterator type that gets a table row by its primary key fields.

In the following code example, a Company Record type is queried from the database using its primary key, "Name".

```
procedure Example()
var Company : Record "Company";
begin
    Company.Get('COMPANYNAME')
...
```

But the following example will also compile and execute without any warning messages – that results in an exception at runtime.

```
procedure Example2()
var Company : Record "Company";
begin
    Company.Get(1,'asdasd',CurrentDateTime(),45)
...
```

In addition, the latter issue introduces new issues:

Issue 5. It is not safe to change table keys.

Issue 6. Lack of parameter hints.

While changing a table key is a very rare occasion, looking up what parameters the Get procedure for a table requires is something that developers will encounter every single day. Since there are more than a thousand tables in the base application, and even more in Client-tailored applications, the AL developer will have to add many unnecessary steps to writing database queries.

There are more examples of such dynamically typed input problem, but some more notable ones are common user interface prompts, such as "StrMenu" or state changes for the Record iterator type, like "CalcFields", which can only be used on *Calculated Fields* – a different type of field, which acquires its value from a calculation formula.

2.4 Code Reuse and Type Safety

Issue 7. Strict type annotations.

As was shown in (Hvitved, 2009), a lot of code duplication in C/AL could be attributed to the lack of polymorphism, as strict type annotations do not allow code reuse between similar table objects. It is still the case today, as tables still do not have support for subtyping.

For example, consider the 3 similar procedures in Listing 25 in Appendix 2 on page 76, that return a list of unique product categories from a sales document, a purchase document and a sales invoice document. Each of these functions strictly take an input parameter of one singular record type (lines 1, 16 and 31).

To avoid such repetitions, AL developers often use the Variant type, which is a dynamic type.

Issue 8. Common use of the Variant type

Dynamic functions in AL are built on Table Id-s and Field Id-s – Integers, that are unique to the specific table or field that is accessed.

But dynamic code some introduces some caveats. While it solves the maintainability problem caused by the need for duplication, there are some issues with it, namely:

- Loss of expressiveness (as it is necessary to track integers)
- Loss of compile time safety checks
- Loss of performance

For example, Listing 26 combines the 3 procedures of Listing 25 into one, but it comes with the cost of drawbacks stated above and having to know the specific Field Id's.

Issue 9. Code duplication on Object types

There are many similar objects in Business Central, which due to restricted code reuse implement the same fields, procedures and patterns.

In (Hvitved, 2009) there are some design patterns as input for a future object-oriented design of the language. Table 2 lists some of these patterns and how many times these patterns are implemented in the current base application (version 19.5) of Business Central.

The objects are counted by the name of the object, e.g. "Item Template" is considered an object of type Template and an "Item Journal Line" is considered an object of the type Journal Line). Abbreviations, such as Jnl. (Journal) are also counted.

Design pattern	Count (Pattern)	Description
Master tables	not counted	Primary/master data such as customers and items - can be thought of as entities
Templates	41 (Template)	Used in connection with journals, describes common control information such as type of journal, numbering series, account numbers etc.
Journals	12 (Journal Line) 10 (Journal Batch) 5 (Jnl. Line) 1 (Jnl. Batch)	Unposted data, i.e. information that may change before it is posted to the ledger
Ledgers	18 (Ledger Entry) 2 (Ledg. Entry) 3 (Entry Buffer)	Posted data, i.e. events that cannot and should not be deleted by law. Used as accounting data.
References	not counted	Data that is always referred not copied e.g. Post Code
Registers	14 (Register)	Used to group entries in ledger tables that belong to the same posting
Posted documents	20	Only used for easier reference to the posted data, used to group ledger entries
Virtual tables	not counted	Tables that do not exist in SQL, e.g. Table Integer contains all integers, Table Date contains all dates

Table 2. NAV Table Design patterns from (Hvitved, 2009) and their current implementation counts.

To illustrate the issue, there are 20 Ledger Entry pattern implementations mentioned in Table 2 (above), and these tables each separately define the fields listed in Table 3. The obvious issue here is that many fields are declared *n* times, once for each table, which expands the size of the codebase and increases the cost of refactoring.

The same kind of duplication exists for procedures as well, e.g. there are 15 declarations of GetLastEntryNo and 14 declarations of ShowDimensions, which all have the same method body.

Field name	**Type [*Length*]**	**Declarations (out of 20)**
Entry No.	Integer	20
Posting Date	Date	18
Document No.	Code[20]	18
Amount	Decimal	16
Description	Text[100]	15
User ID	Code[50]	15
Global Dimension 1 Code	Code[20]	14
Global Dimension 2 Code	Code[20]	14
Dimension Set ID	Integer	14
Document Type	Enum	13
Source Code	Code[10]	13
Journal Batch Name	Code[10]	13
Reason Code	Code[10]	13
Shortcut Dimension 3 Code	Code[20]	13
Shortcut Dimension 4 Code	Code[20]	13
Shortcut Dimension 5 Code	Code[20]	13
Shortcut Dimension 6 Code	Code[20]	13
Shortcut Dimension 7 Code	Code[20]	13
Shortcut Dimension 8 Code	Code[20]	13
External Document No.	Code[35]	12
Amount (LCY)	Decimal	11
Document Date	Date	11
Currency Code	Code[10]	10

Table 3. Duplicate field declarations in Ledger Entry tables.

2.5 Interfaces

For the kind of tasks mentioned in the previous section, modern programming languages provide Interfaces, the use of which facilitates both code reusability and type safety.

The Interface type was added to AL recently, namely in 2020[1], but because of legacy limitations it has one major restriction – Interfaces can only be implemented on Codeunits.

While the current interface type provides easily changeable components to the application, such as price calculation logic or various kinds of barcodes, it does not solve the duplication problem highlighted in the previous section.

As there is no subtyping to build the interface on, the best an interface could do to reduce duplicated code is a procedure with a (dynamic) Variant parameter.

The Interface Line With Price (below) is implemented in exactly this way in the AL base application with the following signature consisting of dynamic parameters:

```
interface "Line With Price"
    procedure GetTableNo(): Integer
    procedure SetLine(PriceType: Enum "Price Type"; Line: Variant)
    procedure SetSources(var NewPriceSourceList: codeunit "Price Source
        List")
    procedure GetLine(var Line: Variant)
    procedure GetLine(var Header: Variant; var Line: Variant)
    ... 10 other procedures
```

Such approach further contributes to the amount of dynamic parameters highlighted in Issue 4.

2.6 Constructors

Issue 10. Constructing objects and assigning fields is unnecessarily verbose

Object types in AL are constructed like in the following example, repeating the name of the constructed object on each line.

[1]https://docs.microsoft.com/en-us/dynamics365-release-plan/2020wave1/dynamics365-business-central/al-interfaces

```
procedure Example()
var
    SalesInvoiceLine: Record "Sales Invoice Line";
begin
    SalesInvoiceLine.Init();
    SalesInvoiceLine.Type := SalesInvoiceLine.Type::Item;
    SalesInvoiceLine."No." := '100100';
    ... rest of the procedure
```

Listing 2. Assigning the fields of a Sales Invoice Line object.

It is currently intentionally like this, as there was a way to set fields without writing the object name for each field using the *with* statement, but that feature is deprecated because of the situation in the following example.

On line 16 in Listing 3, the Name field is not assigned in the Sales Invoice Line object (defined on line 9), but instead in the Company (defined on line 5) object. The reason being that the field names are looked up from multiple places, including recursive *with* statements and the SourceTable expression.

As the field Name does not exist in Sales Invoice Line, the field is assigned for the next object that has Name defined. Thus, the *with* statement was deprecated, to prevent the confusion of recursive definitions and accidental field assignments. For more information on *with* statements refer to the official documentation [1].

```
 1 page 60001 CompanyPage
 2 {
 3     Caption = 'CompanyPage';
 4     PageType = Card;
 5     SourceTable = Company;
 6 
 7     procedure Example()
 8     var
 9         SalesInvoiceLine: Record "Sales Invoice Line";
10     begin
11         SalesInvoiceLine.Init();
12         with SalesInvoiceLine do
13         begin
14             "No." := '100100';
```

[1]https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-deprecating-with-statements-overview

```
            Type := SalesInvoiceLine.Type::Item;
            Name := 'name';
        end;
    end;
}
```

Listing 3. The (obsolete) *with* statement assigning fields without explicit declaration.

2.7 Data Relations

Issue 11. No member access expression for relations

Nested object structures are not supported in AL. To store nested object structures, they are commonly defined in the format of a Table relation, as shown in the following example.

In the following example there a *Record* Sales Header containing many Sales Line *Record*s by a composite key of "Document No." and "Document Type".

```
table 37 "Sales Line"
{...
    fields
    {...
        field(3; "Document No."; Code[20])
        {
            Caption = 'Document No.';
            TableRelation = "Sales Header"."No."
                WHERE("Document Type" = FIELD("Document Type"));
        }
...
```

Listing 4. Related table key definition.

As an alternative way of storing nested object structures, it could also be possible to use XML or JSON objects.

What makes these relations different to a modern language, is that:

- Usually all table relations (i.e. SQL joins) in AL are written by hand
- The developer has to know these relations exist, as these relations are not linked by

a navigation property and therefore lack code-completion

- The programming style of AL makes accessing nested structures verbose

So what would be a single line in a modern programming language:

```
SalesHeader.Lines
```

Is commonly written in the following way in AL:

```
//...first declaring the variable
    SalesLines : Record "Sales Line";
//...in the code
    SalesLine.SetRange("Document No.", SalesHeader."No.");
    SalesLine.SetRange("Document Type", SalesHeader."Document Type");
    SalesLine.FindSet()
```

Of course, a Header table accompanied by a Line table is a very common design pattern in the Business Central database, where the Header is to store information about a document, that should not be duplicated on every single line. Thus, the name of the table itself is indicative of having a Line table.

2.8 Reading and Writing Data

Issue 12. No support for serialization or deserialization.

Another unusual part of the AL language is reading data and writing data from and to external sources. The most common data formats used in Business Central are XML, JSON and XLSX files.

In most modern languages these tasks are handled through serialization and deserialization into a strongly typed data structure. For the reasons stated in Issue 11, deserialization is not possible in AL without writing the deserialization code manually using workarounds.

The reason being that parent Records are not linked to children and doing so manually would bring questionable benefits, as each nested object type will then have to be stored into a separate variable, that will also have to be manually filtered and queried from, as there is no member access navigation. Automatic serialization of non-nested objects is possible and could be implemented using the same functions used for writing dynamic code through field IDs.

In Listing 27 in Appendix 2 on page 76 is an example of how deserialization can be done today. There is a Field Record type defined on line 3, that contains all fields and their respective IDs for a table, which can then be used with the help of the dynamic FieldRef (see: Figure 1) type for JSON serialization.

But because most data structures are nested, reading JSON data in AL is written by hand, making development of web requests time-consuming by requiring explicit type casting and property navigation.

An example of a JSON request can be found in Listing 24, that we will later use for comparison with the proposed improvement.

2.9 Legacy

As the language is old in its core, and has undergone many changes, it has some unexpected behavior. The unexpected behavior may even be depended on in some cases, if the codebase has remained the same for a long time. However, when writing new code, it would be preferable to bypass or avoid such cases.

Issue 13. Legacy code behavior

Bear in mind that many of these quirks are already highlighted by the compiler, but still allowed to not break existing partner code. Here is a non-exhaustive list of examples:

1. Inside *repeat* loops - changing a key property value of the Record also changes the cursor position of the loop, may result in infinite loops if not careful.
2. Procedures may be called with or without brackets, meaning Rec.Insert and Rec.Insert() are equivalent - Version with brackets is preferred as it's safe against field naming collisions.
3. There can be a global and a local variable with the same name - e.g. Event Subscribers consistently use global variables when calling procedures.
4. Legacy *with* statements will implicitly use the SourceTable variable as explained in Issue 10.

5. Certain variable names should be avoided, for example setting the TableNo property for a Codeunit will override the Rec variable and DotNet assemblies can not import a value named Value.
6. A Record parameter passed without a reference cell will not pass existing filters.

Another quirk in the language is inconsistent array indexing. While the majority of the AL language is 1-based, e.g. the function StrPos(1) refers to what a computer scientist would understand as index 0, there are exceptions to this.

For example, when accessing the nodes of XmlNodeList, the following code gets the first child element – using a 1-based index.

```
NodeListVariable.Get(1, TempNode);
```

But when accessing the nodes of JsonArray, the following code will get the second child element – using a 0-based index.

```
JsonArrayVariable.Get(1, TempToken);
```

3 The Requirements

In the current chapter we define the requirements for a viable solution to the issues defined in Chapter 2.

As discussed in Chapter 1 the source language chosen for the compiler is F#, as it has been successfully used for supporting other target languages, for example Javascript[1], Python[2], and PHP[3]. Using a language from the same ecosystem also allows to reuse some tools, that were originally written in C#, and eases type-mapping, as many AL types have .NET equivalents as shown in Figure 1. It also satisfies the domain constraint of using an officially supported .Net language maintained by Microsoft to provide long term perspective to the proposed solution.

3.1 Functional Requirements

Requirement 1. Versioning

As Business Central and AL is regularly updated with new features, an alternative solution must explicitly support specific versions of the AL runtime (see: Chapter 4) to provide maintainability under further updates to AL and Business Central. Such versioning should be supported with minimal manual adjustments, to help with the maintainability of the project.

Requirement 2. Integration with existing AL libraries (i.e. app packages) and providing functionality to libraries written in AL

The AL codebase is massive, and writing code that does not use any existing AL library would severely limit the functionality and viability of a solution, meaning that thousands of object types from AL must be usable in F# as they would in AL. It should also be possible to use the library developed in F# from libraries written in AL.

Requirement 3. The AL core library

[1]Fable (2021). *Fable - JavaScript you can be proud of!* Accessed: 2021-11-23. URL: https://fable.io/.

[2]Fable Python (2021). *Python bindings for Fable*. Accessed: 2021-11-23. URL: https://github.com/fable-compiler/Fable.Python.

[3]Peeble (2021). *Peeble. A F# -> PHP transpiler*. Accessed: 2021-11-24. URL: https://github.com/thinkbeforecoding/peeble.

The developer must be able to write F# code that uses the same functions used in AL, as AL has a large library of domain-specific features, that may need to be used directly from F#.

Requirement 4. Type mapping

The AL runtime and type system is implemented in C# with wrappers around primitive types. A viable solution should map these types to the semantic equivalents automatically, letting the user write code that could (to certain extent) also be tested in F#.

3.2 Requirements for Enhanced Code Quality

Requirement 5. Reduce boilerplate for readability

As shown in the issues 2, 10, 11 and 12, a major goal for readability is to remove unnecessary formalities from the code and reduce noise around relevant code for the developer.

Requirement 6. Type safety

In issues 4, 5, 6, and 8 we showed that there are possibilities for bugs to be introduced due to loosely typed functions and that in certain cases type safety comes at the cost of code duplication. A viable solution should guarantee type safety without such drawbacks.

Requirement 7. Stable and consistent interface around legacy code

As shown in the Issue 13, AL has certain quirks and inconsistencies on the language level that can not be changed as it would break legacy code. An alternative solution should provide a stable and intuitive interface, that handles these cases in the background.

Requirement 8. Introduce code reuse where possible

Issue 9, shows an issue with a lot of similar objects, that have no shared code between them - inheritance or composition would allow for shared architecture and reduce the amount of code required asymptotically.

3.3 Sample End-to-end Use Cases

As the AL language has a large ecosystem for a domain-specific language and providing the full functionality of AL is not attainable in the scope of this thesis, we need to narrow the proposed approach down to a few significant end-to-end use cases, that would be illustrate the effect of the solution immediately.

Use case 1. HTTP requests with JSON

As the entire Business Central platform is moving away from .NET code and .NET extensions will not be supported in the near future, one major point that could be improved is integration with external services, that are used to replace this existing .NET code. It will make it possible to utilize modern F# features like Type Providers and providing autocomplete to the developer, and preventing spelling mistakes and false type casts.

Use case 2. Object-oriented design for Record types

In Issue 9 we showed that Business Central has many design patterns, that are implemented purely by duplication, without language level support. Designing tables with shared fields and procedures via inheritance would greatly increase maintainability and speed of development, also eliminating the possibility for inconsistencies, that may appear with mass duplication of code.

4 Design and Development

In the current chapter we will first go over a high-level overview of the current AL and planned F#-to-AL development cycle, then describe the tools used to develop the compiler and then go into further detail about design and development. The source code of the project will be publicly available on GitHub[1] and some examples will refer to the source code directly.

For further reference, we use the following tools and packages for development:

- AL Language extension version 8.4.8.62398 (i.e. runtime version 8.0)
- Base Application version 19.5.36567.36700
- System Application version 19.5.36567.36700
- Business Central sandbox artifacts for version 19.5.36567.36700

4.1 High-level Overview of Development

AL development is done in Visual Studio Code, using the AL Language[2] extension. The AL development cycle is illustrated in Figure 2 (below),

1. The AL developer downloads symbols (i.e. *app packages*) from a Business Central server.
2. The symbols (contained in a compressed .app file) are extracted and read by Visual Studio Code.
3. The developer now has access to the full AL source code and any dependencies inside the versioned *app package* files for development.
4. The developer compiles the project into another *app package* file.
5. The developer publishes the *app package* to the server, which is then compiled into C# during runtime.

[1]Ian Erik Varatalu (2022). *FSAL - F# to AL compiler*. URL: https://github.com/fsal-compiler/fsal-compiler.

[2]*AL Language* (2022). Accessed: 2022-4-28. URL: https://marketplace.visualstudio.com/items?itemName=ms-dynamics-smb.al.

6. The development environment launches a web browser, opening a page for an AL object specified in a launch.json file.
7. If a debugger is launched, then the developer can now step through breakpoints in AL.

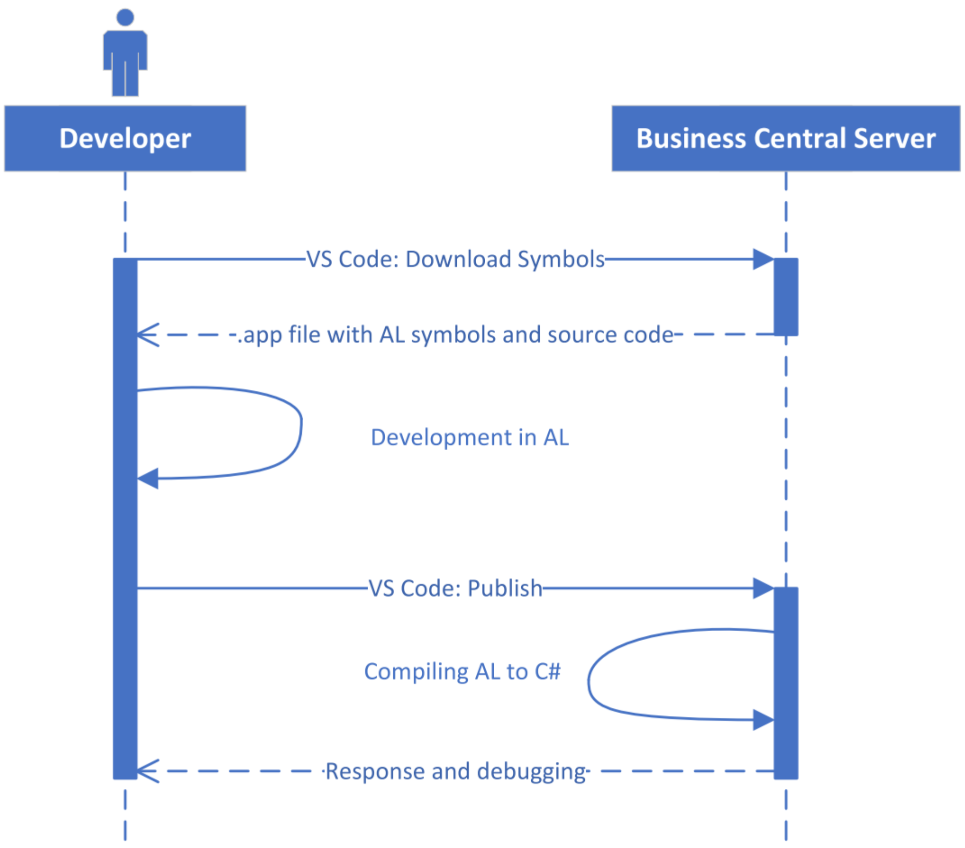


Figure 2. High-level overview of the AL development cycle.

To develop an *app package* in F#, two steps will be added to the development cycle as shown in Figure 3:

1. The downloaded *app package*'s contents must be translated to F# for development
2. The written F# must be translated back into AL before compilation and publishing

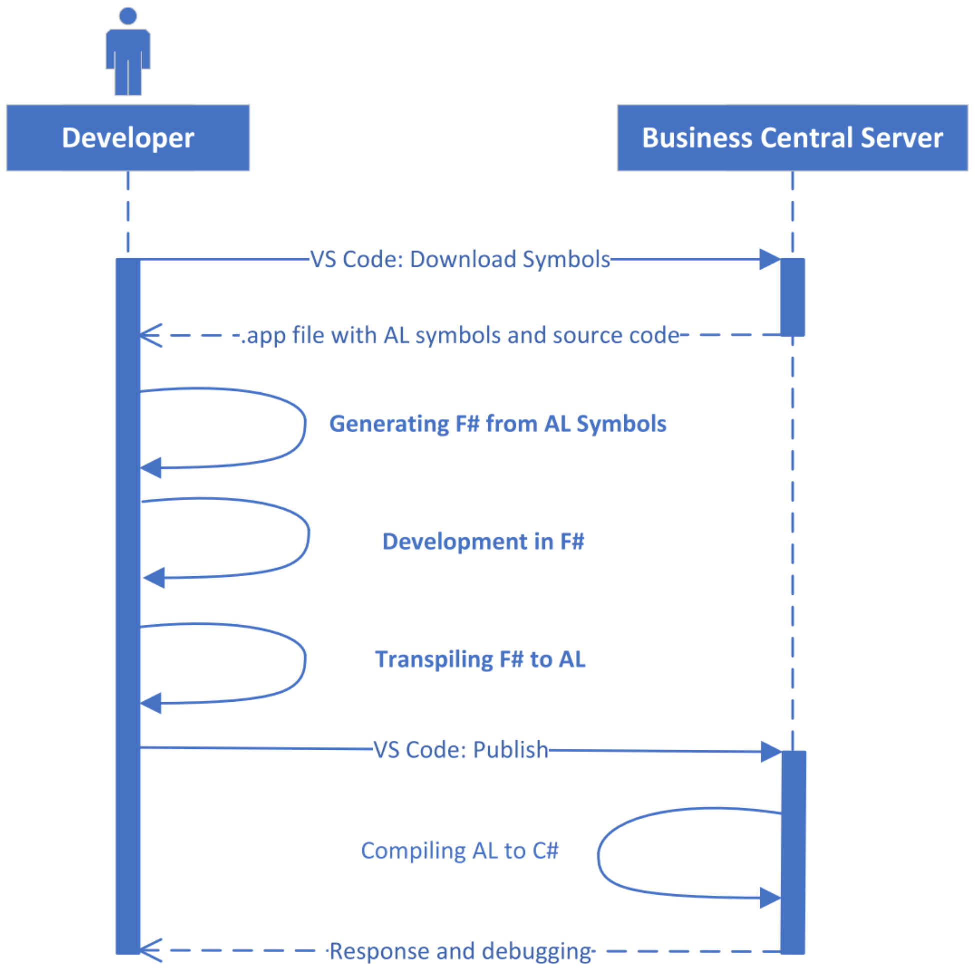


Figure 3. High-level overview of the alternative F# development cycle.

Note that the downloaded *app packages* do not contain all the development information, some built-in types and procedures are provided directly by the AL Language extension.

4.2 Metadata for Objects

As we explained in Requirement 1, versioning the AL language is an important design choice for the compiler. Before going more in depth about the tools chosen for versioning and generating the AL core library, we will go over what is possible.

4.2.1 App Packages

Some versioned components are distributed via app packages, for example the entire *base application* consisting of 1476 Tables, 2686 Pages, 1422 Codeunits and many other types. App packages are also used to distribute the *system application*, that contains essential Business Central components, such as *virtual tables* (see: Table 2) and *system tables* (e.g. File). The rest of the AL Language is versioned via the version of the Business Central runtime.

When defining the manifest of the app package, the developer specifies a runtime version (currently 8.0), that the package is for. Then, the package can be published to Business Central servers, that have the same, or an earlier version (e.g. 9.0) of the runtime.

App packages are the standard way to distribute metadata to the AL Developer. An app package is a compressed archive, that contains the JSON symbols for all objects, procedures, fields and other relevant information. App packages can also optionally contain the AL source code, that can be used for debugging or development. App packages are available for both Cloud and On-Premises versions of Business Central. These app packages are also openly versioned with source code on GitHub[1], where it's easy to track changes.

4.2.2 Business Central Database

Alternatively, a more-detailed XML metadata format exists in the Business Central database, that is stored for each object in an SQL column. In the "Application Object Metadata" table, accessible only on the On-Premises versions of Business Central, there are 3 BLOB type columns for each object, "Metadata" containing XML metadata, "User Code" containing C# code, and "User AL Code" containing the original AL code. Since the Cloud version of Business Central is the primary target for this compiler, the app packages are the better option.

4.3 Metadata for the Runtime

Versioning the runtime functionality is much harder, as there is no publicly available repository or list for types and procedures.

[1]Stefan Maron (2022). *Business Central Code History Repository*. Accessed: 2022-04-30. URL: https://github.com/StefanMaron/MSDyn365BC.Code.History.

Runtime versioned functionality includes:

- All graphical user interface procedures, like menus
- All built-in procedures for objects, like Insert or Delete for records
- List of all built-in types, like HttpClient or JsonValue

4.3.1 The AL Language Extension

To make tools that are compatible with changes to the runtime, many AL development extensions depend directly on the AL Language extension (closed-source), as this allows the extension to work with newer versions that come out. Some examples of this are third-party refactoring tools, analyzers and language-server integrations, like AL XML Documentation[1] and AZ AL Dev Tools[2] extensions. The AL Language extension is also a reliable source of versioning for our compiler.

4.3.2 Business Central Docker Container Artifacts

Microsoft regularly publishes versioned artifacts for every Business Central version to their container registry[3]. A popular development tool, BCContainerHelper[4], can use these artifacts to install any version of Business Central into a Docker container along with other useful data, including AL source code, legacy DLL's, many Powershell scripts and developer tools. The artifacts package also contains a compatible version of the AL Language extension.

4.4 Transformation of App Packages to F#

To import existing AL code, as discussed in Requirements 1 and 2, we need to turn AL app packages into a format, that is usable from F#. For this transformation we use a feature of F# called Type Providers (Syme et al., 2012), that can generate code during

[1]365 business development (2022). *AL XML Documentation*. Accessed: 2022-04-30. URL: https://marketplace.visualstudio.com/items?itemName=365businessdevelopment.bdev-al-xml-doc.

[2]Andrzej Zwierzchowski (2022). *AZ AL Dev Tools/AL Code Outline*. Accessed: 2022-04-30. URL: https://marketplace.visualstudio.com/items?itemName=andrzejzwierzchowski.al-code-outline.

[3]Docker Hub (2022). *Dynamics 365 Business Central Sandbox*. Accessed: 2022-04-30. URL: https://hub.docker.com/_/microsoft-businesscentral-sandbox.

[4]Freddy Kristiansen (2022). *BcContainerHelper*. Accessed: 2022-04-30. URL: https://github.com/microsoft/navcontainerhelper.

design time.

To create a Type Provider, there is a tutorial available at Microsoft Docs[1] and F# Type Provider SDK on GitHub[2].

With the type provider, we can integrate the following steps seamlessly into the development cycle, without having to rely on external conversion tools:

- The developer creates a type provider instance and specifies a path on disk
- The type provider extracts the app package symbols from the specified path and generates F# in the background
- The developer now has full autocomplete support with code generated from the app package

4.4.1 Generative and Erased Types

There are two kinds of provided types - erased types and generative types. Erased types are used for massive schemas, where high performance is preferred over having the exact same runtime representation[3]. Generative types have the benefit of generating real .NET types in the compiled DLL file, meaning that it's possible use inheritance and metaprogramming features such as reflection for further code generation. Erased type providers have an additional use case for providing a simpler interface to code, that compiles into a different syntax, which we will later demonstrate with a third party Type Provider in Section 5.1.

Having real .NET types in the resulting DLL also allows for interoperability with PowerShell using the command *Import-Module*, and C#, using a DLL reference. Another benefit of interoperability, is that AL developers are already accustomed to PowerShell, which is the AL developer's go-to tool for automation, as Business Central configuration requires extensive use of PowerShell.

[1]Microsoft Docs (2022b). *Tutorial: Create a Type Provider.* Accessed: 2022-04-30. URL: https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/type-providers/creating-a-type-provider.

[2]Don Syme (2022). *The F# Type Provider SDK.* Accessed: 2022-04-30. URL: https://github.com/fsprojects/FSharp.TypeProviders.SDK.

[3]Microsoft Docs (2022b). *Tutorial: Create a Type Provider.* Accessed: 2022-04-30. URL: https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/type-providers/creating-a-type-provider.

As the AL Language is close to object-oriented in its design, and thousands of types inherit the same procedures (e.g. all *Records* can be inserted and deleted), we chose generative types, which have certain caveats in regard to performance, that we will discuss later in this chapter.

4.4.2 Creating Strongly Typed Functions via Type Provider

In Issue 4, we showed that some procedures in AL are loosely typed, even though all the necessary information is available during design time. The downloaded app package with symbols (as shown in Figure 3) contains the primary key information for all tables in the application. Using this information, we can generate a strongly typed Get (see: Issue 4) procedure directly from the app package. In Business Central, by convention, the primary key for a table is always the first key defined[1].

In Listing 28, Appendix 2 on page 76, there is a JSON symbols definition extracted from an AL app package for Record Company, that was used as an example in Issue 4. On line 21 are the table fields used for the primary key, which refers to a field "Name". This field "Name" is defined on lines 3 to 16, and specified a type definition Text with maximum length 30 on line 5.

From this information in the symbols, we can generate a strongly typed Get procedure for the Record Company. As the Text type is equivalent to the .NET String type, we will generate a strongly typed Get function for the Company Record, that takes a single input parameter of string. We can also generate this parameter with an explicit name, to show the developer which field is the primary key via a parameter hint.

The use of strongly typed functions will be demonstrated in Section 5.3.

4.4.3 Further Considerations for the Type Provider

Because we chose the less performant generative types in Section 4.4.1 and there are thousands of types that need to be generated, as shown in Section 4.2.1, the code generation process for large packages requires a lot of computation.

Listing 29 in Appendix 3 on page 79 shows an example of the generated code. As shown

[1]Microsoft Docs (2022). URL: https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-table-keys.

in Listing 29, we inherited all the generated records from a common ALRecord class (line 1), from which the records inherit common methods, such as Insert or Delete. The ALRecord itself is defined later in Section 4.5. We also override a get_ObjectId() method, to support using Table Id-s (see: Issue 8). Field Id-s are not yet implemented, but still usable by the *FieldNo* procedure inherited from ALRecord.

In the scope of this thesis, we only generated F# types for tables, which for the base application took about 3 minutes to compile, and resulted in a DLL containing *231 355* lines of (decompiled) code. As this code generation process would slow down the IDE, a better way to use the base application is to first compile it in a separate project and then use the compiled output with a NuGet package or a reference. This way we could use the entire base application without any noticeable slowdowns.

4.5 The AL Runtime Types

To avoid defining the entire core library manually, we considered some options in Section 4.3. We first implemented some essential abstractions for the AL type system manually, shown in Listing 5 below, including a base class for SimpleType, ComplexType and ObjectType (see: Figure 1).

```
[<AbstractClass>]
type ALSimpleValue() = do ()

[<AbstractClass>]
type ALComplexValue() = do ()

[<AbstractClass>]
type ALObjectValue() =
    inherit ALComplexValue()
    abstract member ObjectId : int
```

Listing 5. F# abstractions for the AL type system.

Then, using these abstractions as base classes, we used the .NET Reflection capability, to extract procedure signatures from the assembly, that defines the AL Runtime. An example of the end result can be found on Listing 30 in Appendix 3 on page 79.

The process of extracting signatures is as follows:

1. Load the assembly using .NET *System.Reflection.Assembly.LoadFrom* method.
2. Categorize the types in the assembly by subclasses, into simple types and complex types.
3. Categorize all members of each type into properties and methods.
4. Map the types of methods into their semantic equivalents, e.g. convert Text to String.
5. Generate unimplemented method stubs for all properties and methods, that have the same parameters and return types as in AL.

After the process, it was necessary to make further manual changes, such as inheriting ALRecord from the ALObjectValue type (above), to support the restrictions of the type system and defining an Object Id. Using the procedure above, we generated a total of around 5000 lines of code, which. For more examples see the stubs in the source code directly [1] [2].

The process above could be much further refined into a stable pipeline for each version, but for the scope of this thesis, we found it more attainable to do the final modifications manually. For the same reason, certain parts of the AL Library, requiring heavy modification, were not generated. As our knowledge of the type system and language server is limited, perhaps it's best to consult Microsoft directly on how to approach this, if the process is to be turned into a real pipeline in the future.

4.6 .NET Implementations

As we described in use case 1 in Chapter 3, we consider simplifying development external web requests one of the main use cases for the compiler being developed. To make connecting to external web services more comfortable for the developer, we can implement certain parts of the code generated in the previous section into directly into F#.

For example when testing the connection and output of an Azure Function, the AL development workflow (see: Figure 2) currently has some amount of overhead when making changes. The most common way to develop web services and other repetitive tasks, is to set up a Page object for development, then add an OnPageOpen trigger for this page,

[1]https://github.com/fsal-compiler/fsal-compiler/blob/main/src/Fs.AL.Core/ALSimpleValues.fs
[2]https://github.com/fsal-compiler/fsal-compiler/blob/main/src/Fs.AL.Core/ALComplexValues.fs

with the code being developed, and set it as the launch object after publish. This way, the AL Code that was written will hit a breakpoint as soon as it's published and loaded in the browser.

As this inner-cycle of development still takes some seconds, depending on the app package size, to upload the package, compile and launch, an interactive console could be helpful here. With an interactive console, the developer will have some new options for development, namely:

- Evaluating code line-by-line
- Skipping the overhead of publishing

The optimal way to do this, would be to integrate to the real .NET engine behind Business Central, as this would guarantee the function to have the same effect. As this requires knowledge and access to the engine, which we do not have, we instead, implemented the signatures of the following object types in F# (see source code[1]):

- HttpHeaders
- HttpContent
- HttpRequestMessage
- HttpResponseMessage
- HttpClient

With the signatures implemented, we can test functions line-by-line during development and then compile them into AL. The use of this feature is also demonstrated in Section 5.1.

4.7 The Compiler Implementation

The F# to AL compiler is built on top of the F# Compiler[2]. Before going further into detail, we will explain the high-level overview of the process, to help in understanding we

[1]https : / / github.com / fsal - compiler / fsal - compiler / blob / main / src / Fs.AL.Core / ImplementedBcComplexValues.fs

[2]F# Software Foundation (2022). *F# Compiler Docs*. Accessed: 2022-4-6. URL: https://fsharp.github.io/fsharp-compiler-docs/.

also included a diagram from the compiler documentation in Figure 4.

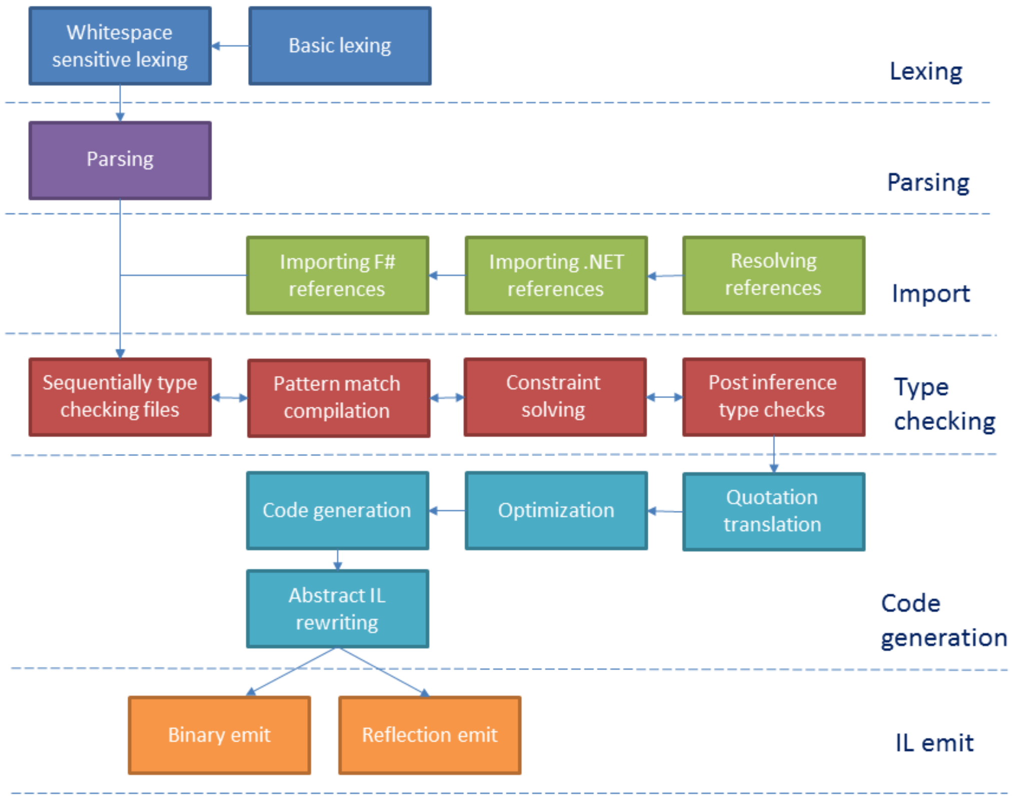


Figure 4. F# Compiler phases - (F# Software Foundation, 2022).

4.7.1 Importing Files and References

The compilation process starts with finding the source code to compile. The F# compilation process is different from AL, as it references all files sequentially, meaning there are no cyclical references between files, and all files used for the compilation of the current file must be declared before, in a top-down fashion.

To maintain this order of compilation, there is a list of included files in a *.fsproj* file, which will be compiled in the exact order that they are specified in. In Listing 6 is the *fsproj* file for the sample project we are using to demonstrate the upcoming compilation process and code samples. Lines 7 to 18 specify the order of files.

```xml
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net6.0-windows</TargetFramework>
    </PropertyGroup>
    <ItemGroup>
        <Compile Include="RecordTest01SimpleRecord.fs" />
        <Compile Include="RecordTest02Inheritance.fs" />
        <Compile Include="CodeunitTest01SimpleMethods.fs" />
        <Compile Include="CodeunitTest02PatternMatches.fs" />
        <Compile Include="CodeunitTest03Constructors.fs" />
        <Compile Include="CodeunitTest04SingleInstanceModule.fs" />
        <Compile Include="CodeunitTest05StrongTypedFunctions.fs" />
        <Compile Include="CodeunitTest06JsonProvider.fs" />
        <Compile Include="CodeunitTest07Legacy.fs" />
        <Compile Include="Demo01HTTPRequests.fs" />
        <Compile Include="Program.fs" />
    </ItemGroup>

    <ItemGroup>
        <PackageReference Include="Fable.JsonProvider" Version="1.0.1"
           />
        <PackageReference Include="Fs.AL.Core" Version="1.0.1" />
        <PackageReference Include="Fs.AL.Packages.BaseApplication"
           Version="19.5.36567.36700" />
        <PackageReference Include="Fs.AL.Packages.System" Version="
           19.0.36528.36625" />
    </ItemGroup>
</Project>
```

Listing 6. The fsproj file for our sample project.

The fsproj file also contains a list of references. For the references in this project, we used NuGet packages, containing the pre-compiled contents of the base application and system packages.

4.7.2 Type Checking

Using the F# files from Listing 6, we create a FSharpProjectOptions data structure specified by the F# Compiler[1], and specify the libraries and (optionally) DLLs used. With the first 3 phases of compilation from Figure 4 handled by the F# Compiler, we can create an instance of a FSharpChecker, that performs the steps in the Type checking phase.

This phase will transform all F# pattern matches[2] into if-else statements before further processing. Additionally, here is where the types of unspecified parameters will be resolved - known as type inference. Pattern matching is an F# feature that ensures all possible cases are handled in conditional branches.

4.7.3 F# Code Generation

In this phase, we can use two more built-in F# compiler features:

1. Optimization. This will replace inefficient code with equivalent expressions, for example replacing unnecessary function calls with the body of the called function, a process known as *inline expansion* (Appel, 1997).

2. Quotation translation. This process is what makes the Type Provider we created in Section 4.4 work. Quotations are a metaprogramming feature in F#, that lets you generate an AST (Abstract Syntax Tree) instead of evaluating code, used to write code for another language, this has been used for example for an F# to OpenCL compiler (Cocco and Cisternino, 2015), to generate optimized GPU code on fly.

For a recap, here's what we have, provided by the F# compiler, that we can transform into compiled AL as well:

- Quotations
- Pattern-matching
- Type inference
- Optimization by the F# compiler (optional)

[1]F# Software Foundation (2022). *F# Compiler Docs*. Accessed: 2022-4-6. URL: https://fsharp.github.io/fsharp-compiler-docs/.

[2]https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/pattern-matching

- Any compile-time constants such as *nameof* expressions or literal strings

4.7.4 Translation to an Intermediate Form

Because F# and AL have very different syntax at times and a single expression in one language may be equivalent to multiple in the other, we transform the syntax tree generated by the compiler to an intermediate form. This section provides a concise overview of the code transformation process, that we will further elaborate on in the upcoming sections.

Using the BNF grammar of AL defined in (Hvitved, 2009) as basis, we define our own syntax, and start reading F# declarations, that we got from the previous phase of the F# Compiler.

First we create our representation of an AL Object in Listing 7 (below), that contains all contextual information that we use when generating AL. This information includes:

1. **F# entity** (line 2), used for quick reference to attributes and interfaces
2. **F# members** (line 3), containing the identifier, arguments and function body for each member
3. **Shared cache** (line 4), this contains abstractions and base types, that we will later use to implement inheritance
4. **AL members** (line 5), this is where we collect information about each AL procedure before code generation
5. **AL fields** (line 6), this contains all contextual information about fields
6. **Object ID** (line 7), the AL object ID
7. **Next field ID** (line 8), this is to auto-generate sequential field ID's for temporary tables, where persistent storage (i.e. having consistent field ID's) is not important

```
type ALObjectBuilder = {
    fsharpEntity : FSharpEntity
    fsharpMembers :  (FSharpMemberOrFunctionOrValue *
       FSharpMemberOrFunctionOrValue list list * FSharpExpr)[]
    sharedCache : SharedBuilderContext
    alMembers : ALMemberBuilder list
    alFields : ALFieldBuilder list
    objectId : int
    nextFieldId : int
}
```

Listing 7. Intermediate data structure for generating AL.

Then, the process of transforming declarations to intermediate form goes as follows:

1. Generate typed AST from all F# files using FSharpChecker.
2. Optimize the syntax tree using the F# Compiler (optional).
3. Select F# declarations that implement one of the base classes we made at Section 4.5.
4. Extract method declarations and group them by Entity symbols (i.e. Types).
5. Set up cache with shared declarations.
6. Resolve the Object ID for all declarations, either by a property or an attribute.
7. Resolve inherited fields for all declarations (implemented in Section 4.12).
8. Generate intermediate data structure for generating AL fields.
9. Extract signatures from declared F# members, i.e. identifiers, parameters and return types.
10. Transform F# member AST to intermediate AST and map function calls to their equivalents.
11. Generate intermediate data structure for generating AL members (i.e. procedures and triggers).

4.8 Language Expressions and Data Structures

The most complex part of compilation is converting the function body to a different language. To stay within a reasonable scope for the thesis, the F#-AL compiler does not yet support many features of the F# language. In this section, we give an overview of the steps taken to convert (relatively simple) F# to AL.

Here are the main steps of converting an F# function to an AL procedure.

- Generating variable declarations from the function body, like shown in Listing 1.
- Transforming F# core library functions to equivalent AL functions.
- Expand F# assignment expressions as there are semantic differences.
- Transforming all member self-references, i.e. removing *this* references from F#.
- Transforming certain F# expressions, like JSON property accesses or pattern matches to multiple AL statements.
- Flattening F# let expressions to a single level, as AL does not support multiple levels.

4.9 AL Code Generation

Instead of generating IL code, like in Figure 4, we will generate AL code. This AL code generation is built using the AL Language extension.

Using the *Microsoft.Dynamics.Nav.CodeAnalysis* assembly, that comes included with the extension, which is used by the language server for linting and refactoring, we can transform our intermediate data structure to real AL code. This ensures that our compiler generates code for the same version of AL and keeps up with changes to the AL runtime.

The mapping functions between our intermediate language and the AL Language extension assembly can be found at https://github.com/fsal-compiler/fsal-compiler/blob/main/src/Fs.AL.Core/ImplementedBcComplexValues.fs

4.10 JSON Type Provider

Using the quotations Section 4.7.3, we can also add support for third party Type Providers. To make development for external web services even more comfortable, we can add Type Provider support for JSON.

Using the strongly typed AST, that we get from Section 4.7.3, we can translate these statements to AL in the background, granting the following possibilities:

- Full autocomplete during development for arbitrary data structures
- Correct type-casting by default without having to do it explicitly (e.g. *AsInteger*)

4.11 F# Modules

Because modules are the most common way to organize F# code, then a developer coming from an F# background may want to use modules. Since there are no static classes in AL, the closest equivalent to a static class is a Codeunit with the property SingleInstance set to true.

To compile F# modules into AL, we introduce an ALSingleInstanceCodeunit (possibly renamed in the future) attribute, where the developer can also specify the Id for the created object. For example, Listing 8 (below) shows a very minimal F# module that compiles into the AL shown in Listing 9.

```
[<ALSingleInstanceCodeunit(60004)>]
module SingleInstanceModule =
    let add2 x = x + 2
```

Listing 8. A F# module that compiles into a single instance Codeunit.

```
codeunit 60004 SingleInstanceModule
{
    SingleInstance=true;
    procedure add2(x: Integer): Integer
    var
    begin
        exit(x + 2);
    end;
}
```

Listing 9. Single instance codeunit compiled from Listing 8.

4.12 Inheritance for Tables

As shown in Issue 9, and (Hvitved, 2009), AL currently has no form of code sharing for Record types. To share code between multiple records we implement inheritance.

Using the shared cache in Listing 7, we pre-process all abstract implementations, and combine the fields and methods into a full object. An example of inheritance is demonstrated in Section 5.2.

4.13 Legacy Mapping

Another possibility we can leverage, as we are not restricted by breaking changes, is changing legacy behavior. For example, we can change all the 1-based indexes of AL to a more common 0-based index. To implement this, we set up mapping functions during compile time, that match functions by their full name in F#. In Listing 10 (below) we convert the 0-based index of the .NET function System.String.Substring to its 1-based index implementation in AL (Listing 11). For more examples of function mapping, see the implementation in source code[1].

We can also provide the functions with context-sensitive tooltips and examples of the differences.

[1]https://github.com/fsal-compiler/fsal-compiler/blob/main/src/Fs.AL.Compiler/IntermediateLanguage/TypeReplacements.fs

```fsharp
[<ALSingleInstanceCodeunit(60007)>]
module Legacy =

    let substrings(content:string) =
        let substring1 = content.Substring(2)
        let someInteger = 6
        let substring2 = content.Substring(someInteger)
        substring2
```

Listing 10. Example Codeunit with 0-based indexes in F#.

```
codeunit 60007 Legacy
{
    SingleInstance=true;
    procedure substrings(content: Text): Text
    var
        substring1: Text;
        someInteger: Integer;
        substring2: Text;
    begin
        substring1 := content.Substring(3);
        someInteger := 6;
        substring2 := content.Substring(1 + someInteger);
        exit(substring2);
    end;
}
```

Listing 11. Codeunit with 1-based indexes compiled from Listing 10.

5 Demonstration and evaluation

In the current chapter we will demonstrate the usage of the compiler in the use cases specified in Chapter 3 and discuss the benefits and drawbacks of the proposed solution. There is also a discussion of features that were out of scope of the current thesis, but which would be beneficial to implement as part of future work.

5.1 HTTP requests with JSON

The first use case for the compiler is easing integration with external services. To fascilitate such integrations, we implemented multiple features that will be demonstrated in this section. We will use the sample AL implementation in Listing 24 in Appendix 2 on page 76 for comparison. The sample originates from the book Mastering Microsoft Dynamics 365 Business Central (Demiliani and Tacconi, 2019) and can also be found online on GitHub[1].

Our demonstration is also available on GitHub[2] with the rest of the source code. Compared to the original implementation, certain functions are put into separate procedures, to point out the required variables from each step of the implementation.

5.1.1 API specification

An example output for the request can be found on the documentation site[3] for the API. We will save this output for use inside a file named api-response.json. Using this example response, we can generate a strongly typed data structure for navigation. For example, the Listing 12 (below) provides us full autocomplete suggestions with types in Figure 5

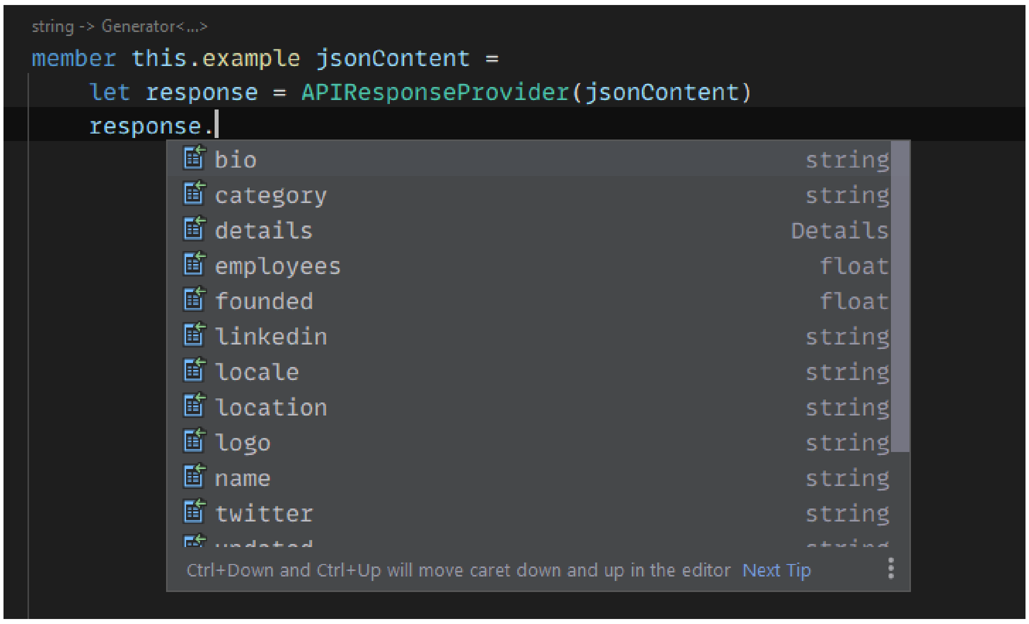
[1]Stefano Demilani (2022). *TranslationManagement.al.* Accessed: 2022-05-05. URL: https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central/blob/master/Chapter%206/TranslateCustomers/TranslationManagement.al.

[2]https://github.com/fsal-compiler/fsal-compiler/blob/main/src/Fs.AL.SampleProject/fsharp/Demo01HTTPRequests.fs

[3]FullContact (2022). *Company Enrich Overview*. Accessed: 2022-05-05. URL: https://docs.fullcontact.com/docs/company-enrich-overview#look-up-company-domain.

```
// generate strongly typed code for the json sample response
let [<Literal>] JsonSample =
    __SOURCE_DIRECTORY__ + "/Samples/api-response.json"
type APIResponseProvider = Fable.JsonProvider.Generator<JsonSample>
```

Listing 12. Generating API spec from JsonProvider.



```
string -> Generator<...>
member this.example jsonContent =
    let response = APIResponseProvider(jsonContent)
    response.
            bio                          string
            category                     string
            details                      Details
            employees                    float
            founded                      float
            linkedin                     string
            locale                       string
            location                     string
            logo                         string
            name                         string
            twitter                      string
            updated                      string
            Ctrl+Down and Ctrl+Up will move caret down and up in the editor  Next Tip
```

Figure 5. Autocomplete with types generated from Listing 12.

5.1.2 Testing Responses

An important part of API integrations is testing sample requests for output. In Section 4.6, we implemented .NET signatures, that would allow us to test requests over HTTP protocol in an interactive console.

For these tests, we implement the request equivalent to our example, as shown in Listing 13 (below). Note that AL *var* variables (i.e. reference cell variables) are declared in F# using the *ref* function and accessed using the Value property. In AL this reference cell does not have to be specified at the call site. The reference cell is used to be fully compatible with all AL reference nuances.

```
// this is to test with replexample.fsx
member this.ProcedureForTesting(input: string) =

    let httpContent = ALHttpContent()
    httpContent.WriteFrom("{\"domain\":\"" + input + "\"}")
    let httpClient = ALHttpClient()
    httpClient.DefaultRequestHeaders.Add("Authorization", "Bearer <YOUR KEY>")
    let httpResponse = ref (ALHttpResponseMessage())
    httpClient.Post("https://api.fullcontact.com/v3/company.enrich",
       httpContent, httpResponse)

    if not httpResponse.Value.IsSuccessStatusCode then
        failwith "Error connecting to the Web Service."
    // just to declare a string in a reference cell
    let responseText = ref ""
    httpResponse.Value.Content.ReadAs(responseText)

    "the response:" + responseText.Value
```

Listing 13. Sample HTTP request for interactive console testing.

Using the above sample, we can now test it directly in the console. In Listing 14 (below) we can evaluate every line of code one-by-one without publishing, which allows us to see changes much quicker compared to the current workflow (see: Figure 2). For example, we can change "testing" on line 5 to something else and immediately re-run it to see differences. This also lets us retain the state of any previous variables, allowing for a speed-up to development similar to Hot Module Replacement[1] used in front-end development.

```
1 // load everything in the project including references
2 #load "./imports.fsx"
3 open Fs.AL.SampleProject.Demo01HTTPRequests
4 let translationManagement = TranslationManagement()
5 translationManagement.ProcedureForTesting "testing"
```

Listing 14. Interactive testing for Listing 13.

5.1.3 Error-handling

In comparison to the AL example used (Listing 24), we could not improve the equivalent error-checking scenarios. If exhaustive error-checking is to be achieved, then it could be implemented using pattern matches, but it was considered to be out of scope for the current thesis.

[1]Webpack (2022). *Hot Module Replacement*. Accessed: 2022-05-05. URL: https://webpack.js.org/concepts/hot-module-replacement/.

Thus, in the F# version in Listing 15 and in the respective compiled form (Listing 16), we implemented error handling in the way suggested in the book (Demiliani and Tacconi, 2019). The getTokenAsObject, getTokenAsArray and getArrayElementAsObject methods are also implemented in the same way in the source code[1]:

```
member this.handleErrors(jContent: ALJsonObject) =
    let details =
        this.getTokenAsObject(jContent, "details", "Invalid response
            from Web Service")
    let locations =
        this.getTokenAsArray(details, "locations", "No locations
            available")
    let location =
        this.getArrayElementAsObject(locations, 0, "Location not
            available")
    ()
```

Listing 15. Equivalent API response error handling to the chosen example.

```
procedure handleErrors(jContent: JsonObject)
var
    details: JsonObject;
    locations: JsonArray;
    location: JsonObject;
begin
    details := getTokenAsObject(jContent,'details','Invalid response
        from Web Service');
    locations := getTokenAsArray(details,'locations','No locations
        available');
    location := getArrayElementAsObject(locations,0,'Location not
        available');
end;
```

Listing 16. Compiled output of Listing 15.

5.1.4 Navigating the Data Structure

As we already checked the errors in the previous section. We can now utilize the generated data structure in Figure 5, and traverse the JSON structure without explicit type casting

[1]https://github.com/fsal-compiler/fsal-compiler/blob/main/src/Fs.AL.SampleProject/fsharp/Demo01HTTPRequests.fs

or intermediate variable declarations.

This means that one expression in F# can be equivalent to multiple statements in AL. For example, the following F# code in Listing 17 compiles into the respective AL code in Listing 18.

Note that the used implementation of JsonProvider[1] is implemented for the Javascript ecosystem and does not allow for error checking done in the same way as the book example[2], as error checking is done by null-checks rather than an if-else statement during parsing.

```fsharp
member this.ReadJsonUsingTypeProvider(json:string,customer:byref<
    Customer>) =
    let response = APIResponseProvider(json)

    let details = response.details
    let location = details.locations.[0]
    let phone = details.phones.[0]

    customer.Name <- response.name
    customer.Address <- location.addressLine1
    customer.''Post Code'' <- location.postalCode
    customer.''Country/Region Code'' <- location.countryCode
    customer.County <- location.country
    customer.''Phone No.'' <- phone.value
```

Listing 17. Navigating JSON in F#.

```
procedure ReadJsonUsingTypeProvider(json: Text; var customer: Record
    Customer)
var
    response: JsonToken;
    details: JsonToken;
    _jtoken: JsonToken;
    location: JsonToken;
    _var4: JsonArray;
    phone: JsonToken;
    _var6: JsonArray;
```

[1]Maxime Mangel (2022). *Fable.Jsonprovider*. Accessed: 2022-05-05. URL: https://github.com/fable-compiler/Fable.JsonProvider.

[2]Stefano Demilani (2022). *TranslationManagement.al*. Accessed: 2022-05-05. URL: https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central/blob/master/Chapter%206/TranslateCustomers/TranslationManagement.al.

```
    _var7: Text;
    _var8: Text;
    _var9: Text;
    _var10: Text;
    _var11: Text;
    _var12: Text;
begin
    response.ReadFrom(json);
    response.SelectToken('details',_jtoken);
    details := _jtoken;
    details.SelectToken('locations',_jtoken);
    _var4 := _jtoken.AsArray();
    _var4.Get(0,_jtoken);
    location := _jtoken;
    details.SelectToken('phones',_jtoken);
    _var6 := _jtoken.AsArray();
    _var6.Get(0,_jtoken);
    phone := _jtoken;
    response.SelectToken('name',_jtoken);
    _var7 := _jtoken.AsValue().AsText();
    customer.Name := _var7;
    location.SelectToken('addressLine1',_jtoken);
    _var8 := _jtoken.AsValue().AsText();
    customer.Address := _var8;
    location.SelectToken('postalCode',_jtoken);
    _var9 := _jtoken.AsValue().AsText();
    customer."Post Code" := _var9;
    location.SelectToken('countryCode',_jtoken);
    _var10 := _jtoken.AsValue().AsText();
    customer."Country/Region Code" := _var10;
    location.SelectToken('country',_jtoken);
    _var11 := _jtoken.AsValue().AsText();
    customer.County := _var11;
    phone.SelectToken('value',_jtoken);
    _var12 := _jtoken.AsValue().AsText();
    customer."Phone No." := _var12;
end;
```

Listing 18. Resulting AL compiled from Listing 17.

5.2 Object-oriented Design for Record Types

For our second use case, we aim to reduce duplicated code, as mentioned in Issue 9. For this purpose we introduced inheritance in Section 4.12. An example of inheritance is demonstrated in the following code sample in Listing 19.

On line 5 of the listing we specify that the type is abstract, meaning it will not compile into a separate record, then we add 3 fields (lines 8 to 11) and a shared function, that all inherited types will implement by default.

Then we declare 2 F# types (lines 14-18 and 21-24), that implement the abstract type. On each of these records we add a single field and specify the object Id, that gets compiled into AL. The compiled AL from these types is shown in Listing 20 and 21 below.

```
 1 // the values will not be initialized in AL
 2 // so we use the default value
 3 let t<'t> = Unchecked.defaultof<'t>
 4
 5 [<AbstractClass>]
 6 type SharedRecordType() =
 7     inherit ALRecord()
 8     member val Id = t<int> with get, set
 9     [<MaxLength(200)>]
10     member val Shared1 = t<string> with get, set
11     member val Shared2 = t<int> with get, set
12     member this.SharedProcedure() = DateTime.Now
13
14 type Inherited1() =
15     inherit SharedRecordType()
16     override this.ObjectId = 60002
17     [<MaxLength(500)>]
18     member val String = t<string> with get, set
19
20
21 type Inherited2() =
22     inherit SharedRecordType()
23     override this.ObjectId = 60003
24     member val Int = t<int> with get, set
```

Listing 19. Inheritance example in F#.

```
table 60002 Inherited1
{
    fields
    {
        field(1;Id;Integer)
        {
        }
        field(2;Shared1;Text[200])
        {
        }
        field(3;Shared2;Integer)
        {
        }
        field(4;String;Text[500])
        {
        }
    }
    procedure SharedProcedure():
        DateTime
    var
    begin
        exit(CurrentDateTime());
    end;

}
```

Listing 20. Result 1 inherited from Listing 19.

```
table 60003 Inherited2
{
    fields
    {
        field(1;Id;Integer)
        {
        }
        field(2;Shared1;Text[200])
        {
        }
        field(3;Shared2;Integer)
        {
        }
        field(4;Int;Integer)
        {
        }
    }
    procedure SharedProcedure():
        DateTime
    var
    begin
        exit(CurrentDateTime());
    end;

}
```

Listing 21. Result 2 inherited from Listing 19.

A full example of this shared-implementation requires prior analysis of object hierarchies, which we have not done in the scope of this thesis, but we have demonstrated that it is possible to mitigate code duplication by inheritance.

Considering the extension oriented architecture of the application, a future implementation should support inheritance for extension types (see: Table 1), which could be used for reuse with modular extensions. Another available solution would be to implement shared fields via composition, as composition would allow to mix-and-match any group of fields or procedures.

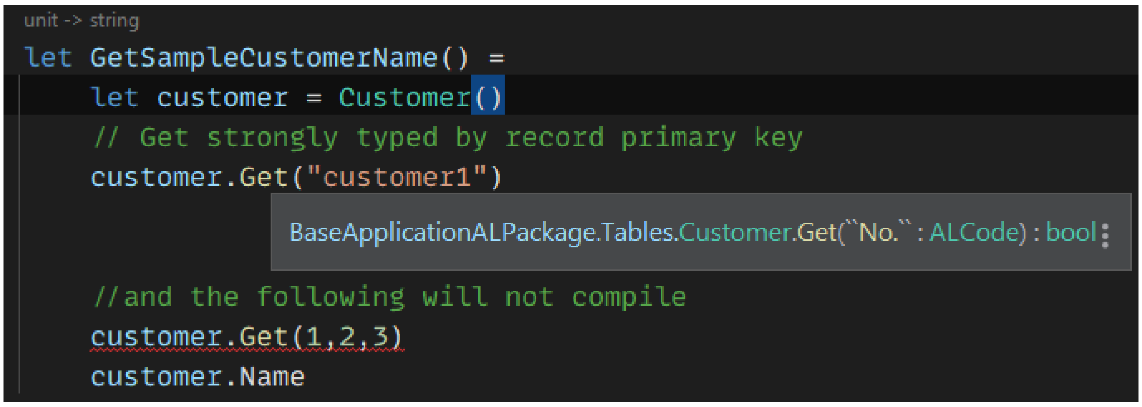
5.3 Additional Features

5.3.1 Strong Typing

In Section 4.4.2 we introduced strongly typed functions, that we can use for enhanced reliability of the resulting AL. The two following figures demonstrate the differences between our strongly typed function and the current implementation.

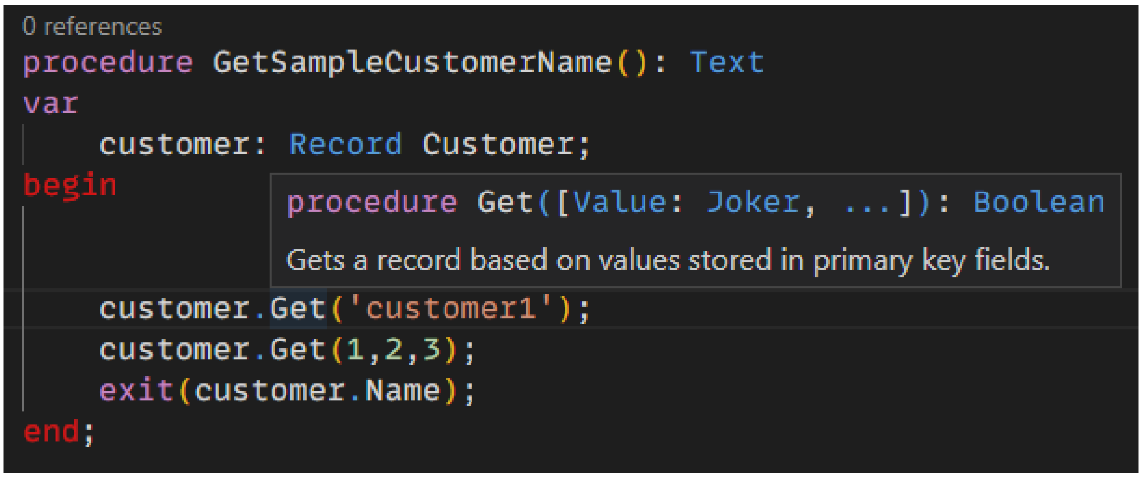
In Figure 6, the developer can use the context-sensitive tooltip to know exactly what the primary key for a Record is, which in our case is a field named "No." of type Code. Additionally, using wrong input parameters will be highlighted by the compiler and raise an error, which means producing the respective AL in Figure 7 from our F# implementation is not possible.

Note that the XML documentation from the AL procedure was not transferred over to the F# library. This limitation is due to the current generation of function signatures in Section 4.5 and will be resolved in the future.



```
unit -> string
let GetSampleCustomerName() =
    let customer = Customer()
    // Get strongly typed by record primary key
    customer.Get("customer1")
                 BaseApplicationALPackage.Tables.Customer.Get(``No.`` : ALCode) : bool
    //and the following will not compile
    customer.Get(1,2,3)
    customer.Name
```

Figure 6. Example of a strongly typed Get procedure in F#.



```
0 references
procedure GetSampleCustomerName(): Text
var
    customer: Record Customer;
begin
                procedure Get([Value: Joker, ...]): Boolean

                Gets a record based on values stored in primary key fields.
    customer.Get('customer1');
    customer.Get(1,2,3);
    exit(customer.Name);
end;
```

Figure 7. Respective AL to example shown in Figure 6.

5.3.2 Constructors

We also implemented F# constructor expressions, that allow the following syntax for concisely assigning multiple properties of a Record similar to the *with* statement shown in Listing 3, but is strictly limited to a single object's fields. This constructor expression and its output are demonstrated with the two following samples.

```
let createEmployee (firstname:string) (lastname:string) =
    Employee(
        ''No.'' = lastname,
        ''First Name'' = firstname,
        ''E-Mail'' = firstname + "." +  lastname + "@business.com",
        ''Phone No.'' = "123123123"
    )
```

```
procedure createEmployee(firstname: Text; lastname: Text): Record
   Employee
var
    returnVal: Record Employee;
begin
    returnVal."No." := lastname;
    returnVal."First Name" := firstname;
    returnVal."E-Mail" := firstname + '.' + lastname + '@business.com';
    returnVal."Phone No." := '123123123';
    exit(returnVal);
end;
```

5.3.3 Pattern-matching

And finally, a feature that the F# compiler translates to if-else statements by default, pattern-matching. As stated in Section 4.8, there are semantic differences between assignment statements in F# and AL, which we handle in the background in the compiler. Illustrated by Listing 22, an identifier can be assinged (line 2) from any expression, which in our case is the match expression spanning lines 3 to 7.

In AL this match expression is expanded into multiple assignment statements (Listing 23) on lines 8, 10, 12, 14 and twice on line 15. Note that AL does support a case statement[1], which can be used similarly to the 3 cases on line 4 in Listing 22. Also note that the if-else cases in Listing 23 form a single 8-line long statement, which ends with a semicolon on line 15.

```
1 member this.patternmatch2 someText =
2     let result =
3         match someText with
4         | "a" | "b" | "c" -> "a, b or c"
5         | v when v.StartsWith "d" -> "starts with d"
6         | v when v.EndsWith "e" -> "ends with e"
7         | _ -> "other text"
8     ALDialog.Message ("matched result:" + result)
```

Listing 22. Example of pattern-matching in F#.

```
 1 procedure patternmatch2(someText: Text)
 2 var
 3     result: Text;
 4     v: Text;
 5 begin
 6     v := someText;
 7     if (someText = 'a') then
 8     result := 'a, b or c' else
 9     if (someText = 'b') then
10     result := 'a, b or c' else
11     if (someText = 'c') then
12     result := 'a, b or c' else
13     if (v.StartsWith('d')) then
14     result := 'starts with d' else
```

[1]https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-al-control-statements

```
    if (v.EndsWith('e')) then result := 'ends with e' else result := '
       other text';
    Dialog.Message('matched result:' + result);
end;
```

Listing 23. Pattern-matches compiled from Listing 22.

5.4 Future Work

As we strived for a working proof of concept, it was necessary to prioritize features to be implemented to make the work feasible timewise. Therefore there are many areas of improvement left for the compiler. As stated in Section 4, the future plan for this compiler is to make it available publicly, and make it a viable alternative, mainly targeted at .NET developers or senior AL developers, as AL retains a big value proposition for being easy to learn for business consultants.

To make the finished product a real viable alternative to AL, the following should be considered:

- Implement missing features. Throughout development, we left many cases unimplemented to save time – A proper implementation would need to consult the full F# specification and either implement or restrict them. For example, we left for loops unimplemented, as they are rare in the language. For more information on missing and edge cases features see Section 5.5.
- Develop a Visual Studio Code extension for the development environment, to ease getting started with an F# to AL project.
- Improve the AL runtime library generated from signatures. The AL library, that we used for this thesis has certain missing features due to lack of knowledge about the language implementation. With some guidance we could implement the full AL library and possibly improve the interactive console as well.
- Implement lambda functions. Lambda functions are a big part of F# and functional programming, which we left out completely. We consulted Microsoft if they plan on implementing delegate types in AL the future, which could be a possibility to bring more F# features to AL.
- Provide support for higher order functions (such as e.g. map and fold). An imple-

mentation depends on how lambda functions will be chosen to be supported.

- Develop AL ControlAddIns (see: Table 1) using Fable[1]. F# already has an JavaScript compiler and AL supports client-side JavaScript integrations, Putting both of these together would create a modern framework for developing client-side extensions for Business Central.
- Consult other F# compiler implementations for design choices. Making a compiler is not a trivial task and throughout developments we noticed many pitfalls. Making adjustments inspired by other implementations would prove very beneficial before extending to the rest of the AL functionality.

5.5 Summary of the Proof of Concept

We currently support only two Object Types: Tables (i.e. Records) and Codeunits. Setting properties other than SingleInstance Codeunits is also not yet supported. Assigning properties for fields (other than MaxLength for Text and Code types) is also not yet supported.

As we implemented features selectively to achieve the use cases shown, following is a list of currently tested and supported features in F# with known edge cases. For more information, a list of working samples and their respective outputs can be found in the source code [2].

- Inheritance
- Type Providers (*assuming functions used in the type provider are implemented)
- Pattern Matching* (not all patterns implemented)
- If-else statements* (not all cases implemented)
- Literal expressions, *nameof* expressions (any compile time constants)
- Constructors* (except certain edge cases with *let* expressions)
- Recursion (no other loops were implemented)
- Mapping certain F# expressions, e.g. *Console.Writeline* compiles into *Dialog.Message* and *String.Substring* will map a 0 based index to 1 (see: Listing 10)

[1]Fable (2021). *Fable - JavaScript you can be proud of!* Accessed: 2021-11-23. URL: https://fable.io/.
[2]https://github.com/fsal-compiler/fsal-compiler/tree/main/src/Fs.AL.SampleProject

- Type-casting for JsonProvider values* (tested with int/string)
- Translating let bindings to variables (and ref/byref types to AL *var* parameters)
- Variable assignments and most operators (excluding "MOD", "+=", "*=", "-=")
- Calling procedures from other object types* (limited implementation, see example in source code)
- Using AL .app files with Type Provider* (currently only compiles AL Records into F#)
- Using procedures derived from the AL runtime* (lack of knowledge about runtime for completeness)

In the current state of development, we managed to support two end-to-end AL development use cases. The F# development environment can use existing AL code via app packages and the resulting AL code can be compiled into its own app package (see: Figure 3) and used from other AL applications as a dependency.

6 Conclusion

In the current thesis we presented a proof of concept compiler implementation for the AL programming language, that provides a mitigation for the 13 issues listed in Chapter 2. The work is an elaboration of the C/AL language redesign idea from Hvitved, 2009.

We started off with highlighting 13 issues in the AL language and development environment provided for Business Central developers that can be improved by utilizing programming language technology available in F#. The issues are illustrated with concrete code examples. The issues were then synthesized into 8 requirements which we started off implementing using F# language and related tools.

Using the created tools, we presented an alternative end-to-end development workflow, that can transform AL to F# and F# back to AL, which allows interoperability with existing AL development.

The first contribution of this work is a Type Provider, which generates F# source code from existing AL code via app packages. Our Type Provider implementation can also transform certain dynamic functions to strongly typed functions, which notify the developer of issues during development. Generating source code this way also lets us version the F# code by the version of the AL app packages and use AL dependencies in F#.

The second contribution is a proof of concept compiler, which is built on the existing F# compiler implementation[1] and translates the code to AL. We also versioned the AL runtime used in the compiler by the Visual Studio Code extension for AL language. This ensures that our solution is compatible with future versions of AL. Using F# as a source language also introduces many new features that could benefit the developer's productivity such as inheritance, JSON deserialization and line-by-line code evaluation.

Type Providers allow our code to have constructs in the language, that compile to a different runtime representation. This feature is what allowed us to implement JSON deserialization in a way that compiles to explicit property navigation, as deserialization to a nested data structure is not currently possible in the AL language.

Our approach allows for fundamental changes to the AL language, while still remaining

[1]F# Software Foundation (2022). *F# Compiler Docs*. Accessed: 2022-4-6. URL: https://fsharp.github.io/fsharp-compiler-docs/.

fully compatible with legacy code. While the current implementation is already usable for the first end-to-end use cases, the functionality could be greatly enhanced and extended to other uses of the platform.

As the AL language will not have an alternative in the near future and the platform has a large ISV (Independent Software Vendor) market with technically complex solutions, the instantiation created in the thesis could fill a missing niche and accelerate the transition to the cloud-based software, that was previously solved by on-premises .NET integrations. As we fulfilled the requirements stated in Chapter 3 with our proof of concept, we conclude that F# can provide a promising modern workflow to current AL development.

The source code of the project is publically available on GitHub[1].

[1]Ian Erik Varatalu (2022). *FSAL - F# to AL compiler.* URL: https://github.com/fsal-compiler/fsal-compiler.

References

Hvitved, Tom (Apr. 2009). “Architectural Analysis of Microsoft Dynamics NAV”. In: Proc. Workshop on 3d Generation ERP Systems, November 2008. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.724.5118&rep=rep1&type=pdf.

Fable (2021). *Fable - JavaScript you can be proud of!* Accessed: 2021-11-23. URL: https://fable.io/.

Brummel, M. et al. (2019). *Programming Microsoft Dynamics 365 Business Central: Build customized business applications with the latest tools in Dynamics 365 Business Central, 6th Edition*. Packt Publishing. ISBN: 9781789131031. URL: https://books.google.ee/books?id=YiyWDwAAQBAJ.

– (2017). *Programming Microsoft Dynamics NAV*. Packt Publishing. ISBN: 9781786468192. URL: https://books.google.ee/books?id=iCVQvgAACAAJ.

Microsoft Docs (2022a). *Data Types and Methods in AL*. en. Accessed: 2022-4-6. URL: https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/methods-auto/library.

Fable Python (2021). *Python bindings for Fable*. Accessed: 2021-11-23. URL: https://github.com/fable-compiler/Fable.Python.

Peeble (2021). *Peeble. A F# -> PHP transpiler*. Accessed: 2021-11-24. URL: https://github.com/thinkbeforecoding/peeble.

Varatalu, Ian Erik (2022). *FSAL - F# to AL compiler*. URL: https://github.com/fsal-compiler/fsal-compiler.

AL Language (2022). Accessed: 2022-4-28. URL: https://marketplace.visualstudio.com/items?itemName=ms-dynamics-smb.al.

Stefan Maron (2022). *Business Central Code History Repository*. Accessed: 2022-04-30. URL: https://github.com/StefanMaron/MSDyn365BC.Code.History.

365 business development (2022). *AL XML Documentation*. Accessed: 2022-04-30. URL: https://marketplace.visualstudio.com/items?itemName=365businessdevelopment.bdev-al-xml-doc.

Andrzej Zwierzchowski (2022). *AZ AL Dev Tools/AL Code Outline*. Accessed: 2022-04-30. URL: https://marketplace.visualstudio.com/items?itemName=andrzejzwierzchowski.al-code-outline.

Docker Hub (2022). *Dynamics 365 Business Central Sandbox*. Accessed: 2022-04-30. URL: https://hub.docker.com/_/microsoft-businesscentral-sandbox.

Freddy Kristiansen (2022). *BcContainerHelper*. Accessed: 2022-04-30. URL: https://github.com/microsoft/navcontainerhelper.

Syme, Don et al. (Sept. 2012). *F#3.0 - Strongly-Typed Language Support for Internet-Scale Information Sources*. Tech. rep. MSR-TR-2012-101. URL: https://www.microsoft.com/en-us/research/publication/f3-0-strongly-typed-language-support-for-internet-scale-information-sources/.

Microsoft Docs (2022b). *Tutorial: Create a Type Provider*. Accessed: 2022-04-30. URL: https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/type-providers/creating-a-type-provider.

Don Syme (2022). *The F# Type Provider SDK*. Accessed: 2022-04-30. URL: https://github.com/fsprojects/FSharp.TypeProviders.SDK.

Docs, Microsoft (2022). URL: https://docs.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-table-keys.

F# Software Foundation (2022). *F# Compiler Docs*. Accessed: 2022-4-6. URL: https://fsharp.github.io/fsharp-compiler-docs/.

Appel, Andrew W. (1997). “Modern Compiler Implementation in ML”. In: Cambridge University Press, pp. 309–343. DOI: 10.1017/CBO9780511811449.

Cocco, Gabriele and Dott Antonio Cisternino (2015). “Homogeneous programming, scheduling and execution on heterogeneous platforms”. PhD thesis.

Demiliani, Stefano and Duilio Tacconi (2019). *Mastering Microsoft Dynamics 365 Business Central: Discover extension development best practices, build advanced ERP integrations, and use DevOps tools*. Birmingham, England: Packt Publishing.

Stefano Demilani (2022). *TranslationManagement.al*. Accessed: 2022-05-05. URL: https://github.com/PacktPublishing/Mastering-Microsoft-Dynamics-365-Business-Central/blob/master/Chapter%206/TranslateCustomers/TranslationManagement.al.

FullContact (2022). *Company Enrich Overview*. Accessed: 2022-05-05. URL: https://docs.fullcontact.com/docs/company-enrich-overview#look-up-company-domain.

Webpack (2022). *Hot Module Replacement*. Accessed: 2022-05-05. URL: https://webpack.js.org/concepts/hot-module-replacement/.

Maxime Mangel (2022). *Fable.Jsonprovider*. Accessed: 2022-05-05. URL: https://github.com/fable-compiler/Fable.JsonProvider.

Appendix 1 – Non-exclusive Licence for Reproduction and Publication of a Graduation Thesis

Appendix 2 – Code Samples

2.1 JSON API Request

```
procedure LookupAddressInfo(Name: Text; var Customer: Record Customer)
var
    Client: HttpClient;
    Content: HttpContent;
    ResponseMessage: HttpResponseMessage;
    Result: Text;
    JContent: JsonObject;
    JDetails: JsonObject;
    JLocations: JsonArray;
    JLocation: JsonObject;
    JPhones: JsonArray;
    JPhone: JsonObject;
begin
    Content.WriteFrom('{domain":"' + Name + '"}');
    Client.DefaultRequestHeaders().Add('Authorization', 'Bearer <YOUR
        KEY>');
    Client.Post('https://api.fullcontact.com/v3/company.enrich',
        Content, ResponseMessage);
    if not ResponseMessage.IsSuccessStatusCode() then
        Error('Error connecting to the Web Service.');
    ResponseMessage.Content().ReadAs(Result);
    //Result containt the ws response
    //Replace <YOUR KEY> with an API key from http://www.fullcontact.
        com
    //Web Services result is a string in json format.

    if not JContent.ReadFrom(Result) then
        Error('Invalid response from Web Service');
    JDetails := GetTokenAsObject(JContent, 'details', 'Invalid response
         from Web Service');
    JLocations := GetTokenAsArray(JDetails, 'locations', 'No locations
        available');
    JLocation := GetArrayElementAsObject(JLocations, 0, 'Location not
        available');
    JPhones := GetTokenAsArray(JDetails, 'phones', '');
    JPhone := GetArrayElementAsObject(JPhones, 0, '');
    Customer.Name := GetTokenAsText(JContent, 'name', '');
    Customer.Address := GetTokenAsText(JLocation, 'addressLine1', '');
    Customer.City := GetTokenAsText(JLocation, 'city', '');
    Customer."Post Code" := GetTokenAsText(JLocation, 'postalCode', '')
        ;
    Customer."Country/Region Code" := GetTokenAsText(JLocation, '
        countryCode', '');
    Customer.County := GetTokenAsText(JLocation, 'country', '');
    Customer."Phone No." := GetTokenAsText(JPhone, 'value', '');
end;
```

Listing 24. JSON API Request from the book Mastering Microsoft Dynamics 365 Business Central.

2.2 Code Duplication

```
procedure GetUsedProductCategories1(Document: Record "Sales Header") :
    List of [Text];
var
    lineIterator : Record "Sales Line";
    categories : List of [Text];
begin
    lineIterator.SetRange("Document No.",Document."No.");
    lineIterator.SetRange("Document Type",Document."Document Type");
    lineIterator.SetRange(Type,lineIterator.Type::Item);
    if lineIterator.FindSet() then repeat
        if not categories.Contains(lineIterator."Item Category Code")
```

```
           then
            categories.Add(lineIterator."Item Category Code");
    until lineIterator.Next() = 0;
    exit(categories);
end;

procedure GetUsedProductCategories2(Document: Record "Purchase Header")
    : List of [Text];
var
    lineIterator : Record "Purchase Line";
    categories : List of [Text];
begin
    lineIterator.SetRange("Document No.",Document."No.");
    lineIterator.SetRange("Document Type",Document."Document Type");
    lineIterator.SetRange(Type,lineIterator.Type::Item);
    if lineIterator.FindSet() then repeat
        if not categories.Contains(lineIterator."Item Category Code")
           then
            categories.Add(lineIterator."Item Category Code");
    until lineIterator.Next() = 0;
    exit(categories);
end;

procedure GetUsedProductCategories3(Document: Record "Sales Invoice
    Header") : List of [Text];
var
    lineIterator : Record "Sales Invoice Line";
    categories : List of [Text];
begin
    lineIterator.SetRange("Document No.",Document."No.");
    lineIterator.SetRange(Type,lineIterator.Type::Item);
    if lineIterator.FindSet() then repeat
        if not categories.Contains(lineIterator."Item Category Code")
           then
            categories.Add(lineIterator."Item Category Code");
    until lineIterator.Next() = 0;
    exit(categories);
end;
```

Listing 25. Duplicated, but strongly typed code.

2.3 Dynamically Typed Code

```
procedure GetUsedProductCategories4(Document: Variant): List of [Text];
var
    dynamicHeaderReference: RecordRef;
    dynamicLinesReference: RecordRef;
    dynamicFieldReference: FieldRef;
    dataTypeManagement: Codeunit "Data Type Management";
    currentCategoryCode: Text;
    categories: List of [Text];
begin
    if not Document.IsRecord() then
        Error('Document is not a Record type');

    // cast variant type to dynamic record type
    dataTypeManagement.GetRecordRef(Document, dynamicHeaderReference);

    // switch case statement based on the Table ID of the dynamic
       record
    // set the table number of the dynamic lines iterator
    case dynamicHeaderReference.RecordId.TableNo() of
        Database::"Sales Header":
            dynamicLinesReference.Open(Database::"Sales Line");
        Database::"Purchase Header":
            dynamicLinesReference.Open(Database::"Purchase Line");
        Database::"Sales Invoice Header":
            dynamicLinesReference.Open(Database::"Sales Invoice Line");
        else
            Error('Invalid Document type %1', Document);
    end;

    case dynamicHeaderReference.RecordId.TableNo() of
        Database::"Sales Header", Database::"Purchase Header":
            begin
                // get the field "Document Type" dynamically by ID
                // this ID happens to be the same for both tables.
                dynamicFieldReference := dynamicLinesReference.Field(1);
                // dynamically constrain "Document Type"
                dynamicFieldReference.SetRange(dynamicHeaderReference.Field(1).Value);
            end;
        // this table doesnt have a document type field
        Database::"Sales Invoice Header":
            begin
            end
    end;

    case dynamicHeaderReference.RecordId.TableNo() of
        Database::"Sales Header", Database::"Purchase Header", Database::"Sales Invoice Header":
            begin
                // get the field "Document No." dynamically by ID
                // this field ID also happens to be the same for all tables
                dynamicFieldReference := dynamicLinesReference.Field(3);
                // dynamically constrain "Document No."
                dynamicFieldReference.SetRange(dynamicHeaderReference.Field(3).Value);
                // set another constraint to only return items
                dynamicFieldReference := dynamicLinesReference.Field(5);
                // this 2 refers to the enumerable value of "Item"
                dynamicFieldReference.SetRange(2);
            end;
    end;

    if dynamicLinesReference.FindSet() then
        repeat
            // field number 5709 is "Item Category Code" for all tables
            currentCategoryCode := dynamicLinesReference.Field(5709).Value;
            if not categories.Contains(currentCategoryCode) then
                categories.Add(currentCategoryCode);
        until dynamicLinesReference.Next() = 0;
    exit(categories);
end;
```
Listing 26. Dynamically typed code combining the functionality of Listing 25.

```
procedure SerializeJson(var recRef: RecordRef) jObject: JsonObject;
var
    field: Record Field;
    fieldRef: FieldRef;
begin
    // finds all fields for a table
    field.SetRange(TableNo, recRef.Number);
    if field.FindSet() then begin
        repeat
            // adds a json property dynamically
            // (cast to a json string for simplicity)
            fieldRef := recRef.Field(field."No.");
            jObject.Add(fieldRef.Name, format(fieldRef.Value));
        until field.Next() = 0;
    end;
    exit(jObject);
end;
```
Listing 27. Usage of the Field table for serialization.

Appendix 3 – Metadata Samples

```
{
    "Fields": [
        {
            "TypeDefinition": {
                "Name": "Text[30]",
                "Temporary": false
            },
            "Properties": [
                {
                    "Value": "Name",
                    "Name": "Caption"
                }
            ],
            "Id": 1,
            "Name": "Name"
        },
        // rest of the fields omitted for brevity
    ]
    "Keys": [
        {
            "FieldNames": [
                "Name"
            ],
            "Properties": [
                {
                    "Value": "1",
                    "Name": "Clustered"
                }
            ],
            "Name": "Key1"
        }
    ],
    "ReferenceSourceFileName":
       ↪ "System%20Tables/Table%20-%20Company.al",
    "Properties": [
        {
            "Value": "Company",
            "Name": "Caption"
        },
        {
            "Value": "0",
            "Name": "DataPerCompany"
        },
        {
            "Value": "Cloud",
            "Name": "Scope"
        },
        {
            "Value": "0",
            "Name": "ReplicateData"
        }
    ],
    "Id": 2000000006,
    "Name": "Company"
},
```

Listing 28. Sample app package symbols for Record Company.

```
public sealed class Company : ALRecord
{
    private string _Name;
    private bool _Evaluation\u0020Company;
    private string _Display\u0020Name;
    private string _Id;
    private string _Business\u0020Profile\u0020Id;

    override int get_ObjectId() => 2000000006;
    public bool Get(string Name) => true;

    [TypeProviderXmlDoc("<summary>Corresponds to property \"Name\" in
       object</summary>")]
    public string Name
    {
    get => this._Name;
    set => this._Name = value;
    }
    // rest of the fields omitted for brevity
```

Listing 29. Sample generated Record Company decompiled into C#.

```
[<AbstractClass>]
type ALRecord() =
    inherit ALObjectValue()
    member this.SetRecFilter ()   : unit = failwith "todo"

    member this.SetPermissionFilter ()   : unit = failwith "todo"

    member this.ClearMarks ()   : unit = failwith "todo"

    member this.Get (values:obj[]) : unit = failwith "todo"

    member this.GetSafe (compilerHashCode:int,values:'t[]) : unit =
       failwith "todo"

    member this.GetBySystemId (systemId:Guid) : bool = failwith "todo"

    member this.SetCurrentKey (fields:int[]) : bool = failwith "todo"

    member this.SetLoadFields (fields:int[]) : bool = failwith "todo"

    member this.AddLoadFields (fields:int[]) : bool = failwith "todo"

    member this.AreFieldsLoaded (fields:int[]) : bool = failwith "todo"

    member this.LoadFields (fields:int[]) : bool = failwith "todo"

    member this.CalcFields (fields:int[]) : bool = failwith "todo"

    member this.SetAutoCalcFields (fields:int[]) : bool = failwith "
       todo"

    member this.CalcSums (fields:int[]) : bool = failwith "todo"

    member this.ChangeCompany ()   : bool = failwith "todo"

    member this.ChangeCompany (companyName:string) : bool = failwith "
       todo"

    member this.Insert ()   : unit = failwith "todo"

    member this.Insert (runApplicationTrigger:bool) : unit = failwith "
       todo"

    member this.Insert (runApplicationTrigger:bool,insertWithSystemId:
       bool) : unit = failwith "todo"
    // 120 other members omitted
```

Listing 30. Sample runtime method stubs for the AL Record type.