Designing and Implementing a Programming Language for Logic

Hans de Nivelle
Department of Computer Science
School of Engineering and Digital Sciences
Nazarbayev University
`hans.denivelle@nu.edu.kz`

May 28, 2020

Project Information
Does this project involve:

(1)	Human Subjects?	**No**
(2)	Animal Subjects?	**No**
(3)	Recombinant DNA?	**No**
(4)	Radiological Materials?	**No**
(5)	Hazardous Materials?	**No**
(6)	Increased Need for Space/Utilities?	**No**
(7)	Commercial Organization/Foundation?	**No**
(8)	Intellectual Property?	**No**
(9)	PI had Social Policy Grant?	**Yes**

1 General Information

Project	
Title:	Designing and Implementing a Programming Language for Logic
Proposed start and final date:	Start 01.01.2021, End 31.12.2023
Duration :	3 years
Location :	Nur-Sultan, Kazakhstan
Requested amount of funding, in USD :	69 684
Principal Investigator	
Name:	Jean Marie (Hans) de Nivelle
Faculty:	School of Engineering and Digital Sciences

2 Project Description

Goal of this project is to design, implement, and to use a programming language that is suitable for implementing algorithms in the field of logic and computer algebra. In addition, we will try to convince others to use it. The resulting programming language will be convenient, both for prototyping and for medium scale projects, and it will be easy to write efficient and maintainable code with it. At the moment of preparing this grant, some parts of this language have been designed in detail, while other parts are still sketchy and need further elaboration. Some important theoretical questions have been solved, while other questions have been hardly looked at. Only a few components have been implemented.

2.1 Motivation

We believe that there is need for a dedicated programming language that supports efficient implementation of logic in theorem proving and interactive theorem proving without compromising code quality. This opinion is based on our own implementation experience, the experience of our project partner Cláudia Nalon, on our teaching experience and on discussions with colleagues.

We have a wide experience with the implementation of theorem provers in C and C^{++} (See [5], [11, 12] and [8]). Theorem provers nearly always consist of two parts: The first part transforms the logic of interest into a normal form. This normal form is much simpler than the logic of interest. Proof search takes place on the normal form. For this part, C and C^{++} are suitable languages. Speed is very important, and the data format is simple.

The second part of the theorem prover performs the transformation into normal form. This part operates on the original logic. For this part, one would like to use higher-level programming language. This language must be efficient on one hand, but on the other hand support high-level programming. We believe that here is a real need for a new programming language. This language must have a type system that is truely helpful when implementing logic. It must detect programming mistakes, and must help with selecting proper function calls. It must be flexible enough so that the programmer is never forced to override it (using casts). The current practice in the field of automated theorem proving is that normal form transformations are also written in C or C^{++}. This is very time consuming, and it results in a situation where it is difficult to extend theorem provers to new logics. We have experienced this by ourselves when we tried to extend our theorem prover Geo ([8]) to 3-valued logic with partial functions. We did not finish the implementation. In general, the theorem proving community is very conservative, still mostly using untyped first-order logic, while refusing to even add simple types. The TPTP language that is the common language for all first-order theorem provers, is untyped (See [21]). We believe that this is for a large part due to the fact that it is very unpleasant to extend theorem provers in C or C^{++}.

The applicant has taught courses on programming paradigms and programming languages at Nazarbayev University. For teaching these courses, we studied OCaml ([14]), Haskell ([13]), Python ([1]), and Prolog ([20]). Before teaching these courses, we expected that some of these languages would solve our problems for implementing logic, but in the end we found that none of these languages met our requirements. We discuss the most important ones in the section on related work.

In addition to the normal form transformation, many theorem provers have third component: One frequently wants to transform the proof back to the logic of interest. We have designed methods for doing this, but never implemented them (See [6], [7]), because we considered it not feasible. With a new, suitable language, implementing these methods may come within reach.

In summary, we want to design a programming language that is optimized for implementation of logic, in which it is possible to write efficient code without compromising good programming style or maintainability. We want to do this together with Cláudia Nalon from the University of Brasília, who experienced the same problems. She is experienced in implementation of provers for modal and temporal logic, which have the same structure and the same problems as first-order theorem provers.

We plan to apply this language to the normal form transformations in theorem provers and to proof transformations of proofs found by these theorem provers. We also aim at other application areas, for example experimenting with new logics in interactive theorem provers or computer algebra. Based on our current insights (especially after teaching Haskell, OCaml and Prolog), we think that it is possible to obtain a language that can compete with all languages that are currently in use for implementing logic.

The key features of our new language are: An expressive type system that is able to express all types that frequently occur in logic. Adaptive typing, which means that the compiler modifies the type of a variable or expression after every operation. This removes the need for explicit type tests (or casts) by the programmer. Carefully controlled side effects. Purely functional programming is the most elegant programming style on the level of small functions, but some crucial parts of a realistic program cannot be effectively implemented in functional languages, namely containers that are intended to store large amount of data. Because of this, we allow methods that modify the state of variables, but in a very controlled version, in order to avoid side effects. As a consequence, it is impossible to create circular data structures, so that there is no need for a complicated run time environment with a garbage collector.

2.2 Completed Parts of Our Programming Language

We explain the parts of our programming language that are already fixed and unlikely to change. We take propositional logic as a guiding example for explaining the difficulties with implementing logic, and how we plan to deal with them. Note that propositional logic is one of the simplest logics possible, much simpler than the logics in [9], [16] or [15] for example.

Definition 2.1 *We assume a countably infinite set of variable symbols. The set of* (propositional) formulas *is recursively defined as follows:*

1. *If p is a propositional variable, then p is a formula.*
2. *If F is a formula, then $\neg F$ is a formula.*
3. *If F_1 and F_2 are formulas, then both $F_1 \rightarrow F_2$ and $F_1 \leftrightarrow F_2$ are formulas.*
4. *If $F_1, \ldots, F_n$ are formulas, then both $F_1 \wedge \cdots \wedge F_n$ and $F_1 \vee \cdots \vee F_n$ are formulas.*

One can see from definition 2.1 that propositional formulas are just trees. If one does not care about types, one can use a programming language with a single tree type and implement all logical algorithms on trees. Some programming languages do this, for example Prolog (See [20]).

This approach becomes problematic in medium sized programs because it leaves all responsibility to the programmer. If the programmer writes a case analysis on the possible forms of a propositional formula, it will be his responsibility to check that all cases are covered. In addition, it becomes very difficult to keep functions for different types of logical formulas apart. For example if one uses first-order logic and modal logic in the same program, both are represented as trees. If one writes some operation on one type of formula, the programming language allows to call it with the wrong type of formula, even when it is meaningless. The compiler or interpreter has no way of finding such mistakes. Such programs may work most of time because different types of logical formulas have much in common, but not always. As a consequence, such mistakes can be hard to find. Even low level programming languages like C have moved away from untyped data structures.

We conclude from the previous that a programming language for logic must have a way of defining types on trees and allowing the same function name to be used for different types. Unfortunately, the type systems of standard programming languages are too rigid for logical formulas. Logical formulas belonging to the same logic can have very different forms, as can be seen from Definition 2.1, and the type systems of languages like C, C^{++}, or Java cannot handle this. As a consequence, the programmer either defines a weak, generic type, or inserts many casts. Both are undesirable.

Functional languages [13] and [14] have much better ways of defining types of logical formulas by means of different constructors. Our mechanism for defining types is similar to the mechanism in these languages, but it is aimed at having a more efficient low level representation, and more flexible. This is Definition 2.1 in the syntax of our language:

Definition 2.2 `struct prop :`
```
   selector sel
   options :                   // Based on sel.
      ?var :
         string p
```

```
?not :
    prop body
?implies, ?equiv :
   prop sub1; prop sub2
?and, ?or :
   repeat: prop sub
```

The selector type can be used to represent logical operators. We do not call it 'operator', because it can also be used for other purposes. In the resulting code, it will be represented by a small integer.

In most cases, our way of defining types is not too different from functional languages, but it has a couple of features that allow more efficient implementation without sacrificing elegance. For example, it is possible to have dynamically growing arrays among the options, which would be not possible in Haskell. Arrays have a fixed prefix, followed by a repeated part, which can grow or shrink like an `std::vector` in C^{++}. The language is designed in such a way that arrays can be modified (member assignment and push_back) in such a way that no side effects are introduced. This is obtained by allowing a restricted form of references. This is more than in Haskell obviously, which is completely functional, but less than in OCaml. Our language allows no side effects, and can be implemented without garbage collector.

The main contribution of our programming language is not in how logical formulas are defined but in how they are used: In order to make the use of formulas easy, we introduce a special kind of subtype called *adjectives*, combined with *adaptive typing*, which means that the context of a formula is used to derive adjectives to the formula. The problem with logical formulas is that their forms can be very diverse. If one has a propositional formula, one does not know in advance which form it has. This causes problems when one wants to access the subparts. A propositional formula can have each of the four forms defined in Definition 2.2. The subformulas `sub1` and `sub2` exist only when `sel == ?implies` or `?equiv`. This is different than most types that occur in other application areas. For example complex numbers always have a real and imaginary part.

If one wants to perform some operation on a logical formula, one has to determine which subtype it belongs to, and only after that one can access the sub structures.

In an object-oriented language (and also Scala) this is done by defining a subtype for every possible way in which the formula can be constructed. One can use dynamic cast in order to determine the type of formula at hand, after which the fields can be accessed. In functional languages (and Scala) one can use matching: Matching simultaneously determines the subtype (or constructor) and assigns the subformulas to local variables.

In our language we follow a novel approach that is both more flexible and more efficient. Firstly, the logical operator is just an ordinary field (`sel` in Definition 2.2). It has type `selector`, which will be represented by an integer constant. Existence of later fields possibly depends on the values of earlier fields.

This is expressed by the `options` keyword. For example in Definition 2.2, the field `body` only exists when field `sel == ?not`. In order to prevent illegal field access while at the same time making legal access easy, we use *adaptive typing.* Adaptive typing means that the same expression can have different types at different points in the program. For this purpose, we use adjectives. The compiler keeps track of which adjectives apply to which expression at which program point. Adjectives can be derived by conditional statements, by construction, by assignment and by user declaration. In the first case, if one checks (either by `if` or by `switch`) that a field has a certain value, then we know that it has this value after the test. In the second case, if we construct a formula with the fields having certain values, we know that they have these values. In the third case, if we assign an expression to a variable, we know that the variable obtains its adjectives from the expression. In the last case, if the programmer declares a variable to meet a certain adjective, then it will meet this adjective. Note that the compiler will check all assignments. There is no way of circumpassing the adjective system. In order to derive the adjectives, we use *abstract interpretation* (See [4, 17]). We developed an adjective system that can be represented by tree automata (See [3]). The adjective system is used for deriving which fields exist, for overload resolution, and for checking completeness of case analyses. In addition, the programmer can define his own adjectives, which can be used for example to define normal forms like negation normal form. Defined adjectives can be used in declarations of variables, after which it will be possible to define dedicated functions for these normal forms, or check completeness of case analysis against the normal form.

2.3 Review of Previous Research

For the project that we are proposing, ‘Review of Previous Research’ means ‘Review of Existing Programming Languages’. There exist many different programming langauges that are traditionally considered suitable for implementation of logic. We discuss the main ones, and why we think that we can do better. We base our discussion on Definition 2.1 and Definition 2.2.

2.3.1 Haskell

The main differences with Haskell is that we will allow a limited amount of side effects, so that it is possible to assign to array members. Array assignment takes constant time when the array is not shared. It is possible to implement containers in such a way that this never happens. Secondly, we allow overloaded identifiers while Haskell doesn’t. In Haskell constructors of logical types behave like usual functions that cannot be overloaded. In Haskell, Definition 2.1 has the following form:

```
data Prop =
   Var String | Not Prop
   | Implies Prop Prop | Equiv Prop Prop
   | And [ Prop ] | Or [ Prop ]    deriving ( Show, Read )
```

The identifiers 'Var', 'Not', 'Implies', 'Equiv', 'And', and 'Or' are the constructors. If one wants to construct a formula, one has to write the constructors explicitly, e.g. `And [ Not ( Var ( "p1" )), Var "p2" ]`. In our language, all constructors of type prop are called prop, and one can simply write `prop( ?and, prop( ?not, prop( "p1" )), prop( "p2" ))`. This is because the function prop can be overloaded. In future versions, it may be possible to write `prop( ?and, ( ?not, "p1" ), "p2" )` or `?and( ?not( "p1" ), "p2" )`. The latter can be obtained by means of C^{++} style operator overloading. In Haskell, the constructors 'Var', 'Not', 'Implies', 'Equiv', 'And', and 'Or' will conflict with other constructors of possible other types of formulas, e.g. first-order or modal formulas. In our language, the constructors do not conflict.

In Haskell (and other functional languages including [18]), the main mechanism for inspecting structure is *matching*. Although it is very elegant, it is very rigid also. For example, it will be difficult to group 'And' and 'Or' into a single matching category. In our language, it is straightforward to combine `?and` and `?or` into single condition after which adaptive typing will establish that the `sub` field exists.

2.3.2 Scala

Definition 2.1 would take the form below in Scala. We follow listing 15.16 in [18].

```
sealed abstract class Prop
case class Var( name : String ) extends Prop
case class Not( body : Prop ) extends Prop
case class ImplEquiv( operator : String, sub1 : Prop, sub2 : Prop )
         extends Prop
case class AndOr( operator : String, Array[Prop] sub )
   extends Prop
```

It can be seen immediately that Scala's type has problems dealing with this definition. In the last two classes `ImplEquiv` and `AndOr`, the `operator` should have only two values (`"implies"`/`"equiv"` for `ImplEquiv` and `"and"`/`"or"` for `AndOr`), but the type system of Scala is unable to enforce this. It is also unable to enforce that only those two strings will be used in matching patterns. As a consequence, it is unable to enforce completeness of patterns in matching. Also note that the keyword `sealed` is unable to do its job, because the programmer can freely add different operator strings. It would be better to separate the last two cases into:

```
   case class Implies( sub1 : Prop, sub2 : Prop ) extends Prop
   case class Equiv( sub1 : Prop, sub2 : Prop ) extends Prop
   case class And( Array[ Prop ] sub ) extends Prop
   case class Or( Array[ Prop ] sub ) extends Prop
```

But now the classes `Implies/Equiv` and `And/Or` have different types. This leads to a new problem: Now the cases cannot be combined any more in matching.

This is the reason why the authors of [18] chose the first approach. Refining the hierarchy will not solve this problem.

2.3.3 Prolog

Prolog is a logic programming language, based on SLD resolution. It is a very nice language for prototyping. Its main control mechanism is matching (unification). Matching is very flexible because of the `=..` operator. Unfortunately, Prolog is completely untyped, so that it suffers from all the problems that we listed in the introduction: There is no way to check completeness of cases, and no way to perform overload resolution. Using matching as main control mechanism can be quite inefficient.

2.4 Requirements on Our Proposed Language

In this section, we list the requirements on our proposed language, based on the discussion of other languages above, and the introduction:

1. The proposed language must have a flexible, static type system. The advantage of a static type system is that operations that cannot be correct are automatically found by the compiler, without need for testing. In our experience, this is especially important when small changes are made in an existing program. Without type checking, all parts of the code must be rerun after every change, which can be very tedious. With type checking, errors introduced by small changes are found by the compiler at once.

   The type system must be flexible enough to deal with the complicated recursive structure of logical formulas. Otherwise, programmers will try to circumvent it, either by using casts, or by using the one-type-for-everything approach.

2. The language must support name overloading. Name overloading means that functions with the same name can be defined for different types. The compiler picks the version that fits to the given type. Without overloading, the programmer has to write the name of the type as part of the function name. This is annoying for operations that frequently occur on many types, like e.g. printing. Instead of calling all print functions `print`, the programmer has to write `printstring`, `printformula`, or `printterm`. In logic, this situation frequently occurs because the same operation has to be applied on different logics.

3. The language must support *adaptive typing*, which means that it is able to use conditions and assignments to derive additional information that will be used in type checking and overload resolution. For example in Definition 2.1, existence of field `body` depends on field sel having value `?not`. The statement `if f.sel == ?not then print( f.body )` must pass type checking, because the compiler knows that `f.sel == ?not` at the point where `f.body` is printed. Similarly,

`if f.sel == g.sel && f.sel == ?not then print( g.body )` must pass type checking. We expect that the combination of `switch` with adaptive typing will be as convenient for the programmer as matching, while offering more flexibility, better checking, and more efficiency. Adaptive typing can be obtained by means of *abstract interpretation.* (See [17], [4])

4. The language must have an easy way of remembering and modifying the state of containers and objects with permanent state. These are mostly input/output files and look up tables. The functional paradigm is elegant for small functions operating on logical formulas, but it has problems on bigger programs, because data structures (dictionaries) cannot be naturally represented. This leads to efficiency problems, and restricts the use of functional languages.

5. The language must be efficient. This does not mean 'it must be possible to write efficient programs', it means: 'well-written, maintainable programs are automatically close to optimal'.

6. The compiler must be easy to install under different operating systems. It must be possible to combine programs with C or C^{++}, so that the language can be used for implementing normal form transformations in theorem provers. Completed programs must run natively (without need to install an additional interpreter or run time environment). We will ensure this by writing the compiler in C^{++} and compiling into C. These languages are available on every operating system.

2.5 Expected Impact

We had many discussions with international colleagues about writing theorem provers for non-standard logics or higher-order logics, and the proper programming language for doing this. We had these discussions because we asked for advice with the implementation of [9]. The outcome of these discussions was that most colleagues agree that the proper language for implementation of logic has not yet been found. One needs a language that on one side is efficient enough to replace C or C^{++}, but at the same time allows for truly high-level programming. We hope that some of these colleagues will eventually become users of our language. If yes, it will have impact on many areas where logics are used, varying from verification to social sciences.

Because of similarity of the data structures involved, we expect the language to be usable for computer algebra as well. We have some experience with computer algebra in a student project, in which the student implemented Buchberger's algorithm (See [2]). We will try to convince our colleagues to use our programming language.

3 Methodology and Ethical Considerations

We plan follow an agile approach, which means that we will put emphasis on implementation, sometimes implementing things before their theory is fully developed. It is important to experiment with parts of the language as soon as possible, in order to verify that these parts are well-designed. We expect to frequently reimplement parts, when it turns out that the original implementation does not meet the intended goals. This does not mean that we will neglect specification, it just means that sometimes it is better to implement first, and specify afterwards.

We give the list of subgoals. It is likely that some of the subgoals have to be carried out more than once. For example, if the language will be extended by templates, the syntax will change and the parser has to be reimplemented.

ADJ Determine the optimal adjective system. The adjective system must be sufficiently expressive so that all naturally occurring preconditions can be expressed in it. At the same time it must be fully automatic. The compiler will use it for checking automatically that preconditions of operations are met. We expect that this can be obtained by means of tree automata [3, 10].

SYNT Define a syntax, and implement a parser for it. At this moment, we prefer a syntax similar to Python, with obligatory indentation. Some collegues have expressed other opinions, so we will try out different syntaxes.

RUNTIME1 Implement a first, simple runtime environment. In this runtime environment, all data are represented by Prolog-style lists over the primitive types. This runtime environment is easy to implement, but inefficient when dealing with arrays, because lookup takes linear time. Still, we expect this implementation to be usable. Apart from efficiency, it meets all requirements.

RUNTIME2 Implement the final, optimal run time environment. The main difference with the simplified run time environment is that arrays are represented by continuous blocks of memory, with unused empty space to allow for growth, similar to `std::vector` in C^{++}, or `ArrayList` in Java. Note that the runtime environment is not an interpreter. It is just a library of helper functions.

TEMPL Add templates to the language. Templates are essential for the implementation of data structures, like for example lists, search trees, or hash functions. We give an example of Prolog-style lists:

```
template< typename D >
   struct list :
      selector sel
      options :
         ?nil :    /* no fields */
```

```
        ?cons :
           D first;  list<D> rest
```

In theoretical sense, there is nothing problematic about templates. All one has to do is substitute the paramater D by a concrete type, for example $D :=$ integer, and after that the template becomes concrete code that can be compiled.

On the practical side, templates are very complicated. The biggest problem is: How and where do we specify the conditions on the template parameters in the template definition? In principle, one can postpone all type checking until the moment of instantiation. C^{++} has lived with this solution for 20 years. This approach is undesirable, because it results in error messages that are hard to grasp, and the programmer who receives the error messages is usually not the programmer who wrote the template. Checking before instantiation is better, but then one needs to design the framework for specifying conditions on template parameters. This is non-trivial. The developers of C^{++} have been trying to do this for 20 years, with only partial success (See [19]). In C^{++}-20, an incomplete version has been accepted. We expect that for our language, the problem will be easier because C^{++} has many low level conditions that we do not have in our language.

POLY Study the usefulness of adding run time polymorphism. If we conclude that it is needed, decide in which form. Run time polymorphism means that a variable can have different types, and the concrete type of a value is known only at run time. It is traditionally realized by allowing subclasses, but there are other ways to implement it. Our runtime model can be easily extended with run time polymorphism between compound types, because all compound types are represented by pointers. Runtime polymorphism is much easier to realize than templates. One needs to add a type tag to the values and perform a case analysis on this type tag whenever a polymorphic function is called.

USE Use the programming language in real projects, concretely on an interactive proof checker for logic with partial functions, on the normal form transformation in [9] and on the normal form transformation of a theorem prover for temporal logic.

3.1 Harmonogram

year	scheduled subtasks			
1	SYNT	ADJ	RUNTIME1	
2	RUNTIME2	TEMPL	POLY	USE
3	TEMPL	USE		

3.2 Ethical Considerations

We do not see any ethical risks in this project. We have high academic standards concerning citation and credits. We will give proper credit to students contributions.

4 Budget Plan and Budget Justification

- We want two laptops, one to lend to students and one for the PI. Requirements on these laptops are that it must be possible to write papers on them, and to program in C/C^{++}. Hence it must be possible to install Linux on them, and the screen must not be too small. We estimate that the price of such a laptop is 1 100 USD, so we are requesting 2 200 USD in total.
- We want one desktop PC, in order to have an additional working place in the room of the PI. We expect that the complete device costs 1 000 USD together with a screen.
- All software that we need is in public domain, hence we are not requesting a software budget.
- We want to be able to pay master students and graduate students for training. We want local master and baccalaureate students, and to invest in their training.

kind of candidate	number	months	total cost
NU baccalaureate	2	27	5 400
NU baccalaureate	2	9	5 400
NU master	1	30	15 000

- We need a project manager for 36 months with a total cost of $36 \times 200 =$ 7 200.
- We want to visit our project partner Cláudia Nalon, University of Brasília and that she is able to visit us during the project. We want to have three such visits, direction is not yet determined. It is hard to estimate the cost of a ticket in the present situation. We expect that the total cost of such a visit is 3 000 USD, which creates a total of 9 000 USD.
- We want to be able to invite six guests from Europe or USA for short visits, cost 1 500 USD per guest visit, total cost 9 000 USD.
- We want to be able to attend one conference per year, and we want to be able to send our bachelor/master degree holders to conferences. Our experience from working with students at previous institutes is that the possibility to visit conferences is a strong motivator for students to finish papers.

Most interesting conferences are in Europe or the USA. From our experience a conference visit in total costs on average 2 000 USD. We therefore estimate 6 times 2 000 USD = 12 000 USD.

5 Research Group

The applicant is confident that he as PI possesses the skills that are needed to carry out this project. He has implemented three theorem provers, ([5], [12]), and ([22]). He has implemented interactive proof assistants in Prolog and in C^{++}.

The applicant has good connections to leading researchers working in implementation of logic (Stephan Schulz, Stuttgart, author of the E-prover, Jens Otten, Oslo, author of the CoP family of provers, and Andrei Popescu, Middlesex (London), contributor to Isabelle prover.)

The applicant has taught compiler construction to graduate students at Wrocław University in Poland. These courses ended with student projects, in which the students wrote their own compilers for languages that they designed by themselves. The applicant has provided the students with data structures for the representation of abstract syntax trees (AST) and with a parsing tool.

The applicant has taught courses Programming Paradigms (CSCI 353) and Programming Languages (CSCI 235) in Nur-Sultan. The applicant is familiar with a wide range of programming languages, like Haskell, OCaml, Python, Prolog, Scala, C, C^{++}, and Java.

The international parter (Cláudia Nalon) is a leading member of the TABLEAUX community. She has organized TABLEAUX/FroCoS 2017 in Brasília. She has published on theorem proving in modal and temporal logics, and implemented a theorem prover for modal logic. She taught courses on logic programming and compiler construction. Her CV is attached to the proposal.

We understand that training of local Kazakh people is an important aspect of FDCRGP grants, and we aim to include undergraduate and master students early in the project. We accept a certain risk that these students will be mostly learning from the project, and not so much contributing, especially in the early phase. We are committed to training them. We will require from them that they study different competing programming languages in order to learn from them, and we will require that they reimplement the same logic in different languages, in order to learn and to compare. We will require that they perform literature studies. We may give them tasks that are loosely related to the main goals of the project, like implementing alternative syntaxes, alternative run time environments, or different type systems.

Two students have been already involved in this research in the form of summer internships. The first student succesfully implemented Buchberger's algorithm ([2]), and the second student compared different methods of implementing basic logical algorithms (testing for α-equivalence, and substitution) in C^{++}. After this project, we concluded that C^{++} is not suitable for implementation of logic.

We expect that the project will create interesting projects for master theses, with a good mixture of theory and implementation.

6 Research Environment and Training for Scientists

As explained in the previous section, we plan to include NU students on the master and bachelor level early in the project. Especially in the beginning of the project, we will give them tasks that are aimed at their training rather than at the final goal of the project. We will give them opportunity to travel to international conferences. We have planned for this in the budget.

7 Expected Results

At the end of this project we expect to have a working compiler as well as useful programs that have been developed with it. We expect publications (in addition to [8] which will hopefully be accepted). We expect that there will be master thesis projects, and we may use the new programming language in courses.

References

[1] Various Authors. Python home page. `https://www.python.org/`.

[2] Bruno Buchberger. A theoretical basis for the reduction of polynomials to canonical forms. *ACM SIGSAM Bulletin*, 10(3):19–29, 1976.

[3] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications.* 2007. release October, 12th 2007.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattic model for static analysis of programs by construction or approximation of fixpoints. In Geoff Sutcliffe and Andrei Voronkov, editors, *POPL*, pages 238–252. ACM Press, 1977.

[5] Hans de Nivelle. Description of bliksem 1.00. Can be obtained from `http://tptp.cs.miami.edu/CASC/15/SystemDescriptions.html`, 1998. (A one page description of Bliksem 1.00 for CASC-15).

[6] Hans de Nivelle. Extraction of proofs from the clausal normal form transformation. In Julian Bradfield, editor, *Proceedings of the 16th International Workshop on Computer Science Logic (CSL 2002)*, volume 2471 of *Lecture Notes in Artificial Intelligence*, pages 584–598, Edinburgh, Scotland, UK, September 2002. Springer Verlag.

[7] Hans de Nivelle. Translation of resolution proofs into short first-order proofs without choice axioms. *Information and Computation, Special Issue on the 19-th International Conference on Automated Deduction (CADE-19)*, 199(1):24–54, April 2005.

[8] Hans de Nivelle. theorem prover Geo III. Can be obtained from `http://www.ii.uni.wroc.pl/~nivelle/software/` or from `http://www.tptp.org/CASC/`, 2015-2019.

[9] Hans de Nivelle. Theorem proving for classical logic with partial functions by reduction to Kleene logic. *Journal of Logic and Computation*, 27(2):509–548, 2017.

[10] Hans de Nivelle. Deciding logical relations between inductive types using monadic Horn clauses. Submitted to the Workshop on Practical Aspects of Automated Reasoning, 2020.

[11] Hans de Nivelle and Jia Meng. Geometric resolution: A proof procedure based on finite model search. In John Harrison, Ulrich Furbach, and Natarajan Shankar, editors, *International Joint Conference on Automated Reasoning 2006*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 303–317, Seattle, USA, August 2006. Springer.

[12] Hans de Nivelle and Jia Meng. theorem prover Geo 2007f. Can be obtained from my homepage, September 2007.

[13] Graham Hutton. *Programming in Haskell (second edition)*. Cambridge University Press, 2016.

[14] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml (functional programming for the masses)*. O'Reilly, 2013.

[15] Cláudia Nalon and Dirk Pattinson. A resolution-based calculus for preferential logics. In Didier Galmiche, Stephan Schulz, and Robert Sebastiani, editors, *Proc. of the 9th International Joint Conference on Automated Reasoning (IJCAR) 2018*, volume 10900, pages 498–515. Springer Verlag, 2018.

[16] Cláudia Nalon, Lan Zhang, Clare Dixon, and Ullrich Hustadt. A resolution-based calculus for coalition logic. *Journal of Logic and Computation*, 24(4):883–917, 2014.

[17] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 2005.

[18] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala Third Edition*. Artima Press, 2007-2016.

[19] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006, pages 295–308. ACM, 2006.

[20] Leon Sterling and Ehud Shapiro. *The Art of Prolog (Advanced Programming Techniques).* The MIT Press, 1994.

[21] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[22] Geoff Sutcliffe. The CADE ATP system competition. `http://www.cs.miami.edu/~tptp/CASC/25/`, August 2015.

8 Ongoing Projects

The PI holds a social policy grant on the same topic, with title 'Designing and Implementing a Programming Language for Logic and Computer Algebra'. This grant started on 12.08.2019, and was originally scheduled to end on 12.08.2020. It is currently extended until 31.12.2020, due to the COVID-19 situation.

The PI is frequently invited to PCs of international conferences and workshop, four per year on average. This takes some of his time, but has the advantage that he knows what is going on in the field of automated deduction.

PI expects to be able to spend approximately 40% to his time to this grant. The main uncertain factor is teaching load which can vary greatly at NU and is unpredictable.