Unique Identifiers in Pure Functional Languages ⋆

Péter Diviánszky

Eötvös Loránd Univ., Prog. Lang. and Compilers Dep., Budapest
divip@aszt.inf.elte.hu

Abstract

It is hard to use properties of memory locations in functional languages. One such property is that memory locations are unique identifiers of objects, i.e., two objects are equal if they have the same memory address. I introduce a syntactic sugar called `new` in a pure functional language. `new` provides easy-to-use unique identifiers for expressions. I show moreover that if we use graph rewriting during the compilation, compilation of `new` can be optimised such that the final imperative code use memory locations for identification. Finally I give a prototype implementation of the language in Clean with ABC code.
The introduced syntactic sugar can be expressed in the ν-calculus. The difference is that identifiers are hidden by `new` and the usage of `new` is forced to be more explicit by a so-called new-propagation rule. Due to this two restrictions, `new` fits better in functional programming than the ν-calculus.

1 INTRODUCTION

Definition 1. (Informal) *Identifiers* are values of an abstract data type `Id` together with an identity function `equal :: Id -> Id -> Bool`, and some other functions to generate identifiers. Identifiers can be bound to expressions.

Definition 2. (Informal) *Unique identifiers* are defined by the following property: if the identifiers of two expressions are equal, then the expressions have the same observable behaviour.

The benefit of unique identifiers are well-known: One can distinguish and compare expressions by identifiers.

In imperative languages memory locations of objects are unique identifiers of them. The syntactic sugar `new` defined in this paper offers easy-to-use unique identifiers. The compilation of `new` can be optimised. The optimised compilation results imperative code which use memory locations as unique identifiers.

1.1 The Idea Behind the Syntactic Sugar

Suppose a list data structure with constructors whose last arguments are identifiers:

```
:: MyList a = MyCons a (MyList a) Id
            | MyNil               Id
```

⋆Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. T037742.

Suppose that an equality function is defined on `MyList` as

```
getId :: MyList a -> Id
getId (MyCons _ _ i) = i
getId (MyNil i)      = i

(===) :: MyList a -> MyList a -> Bool
(===) a b = equal (getId a) (getId b)
```

If `y` was defined as `MyCons 3 (MyNil id) id`, the unique identifier property would not hold because `y === tl y` ⇓ `True`. (⇓ denotes the reduction to head normal form.) Such misuse can not happen if the management of identifiers is carried over by a syntactic sugar: Instead of `MyCons 3 x id` one should write `new (MyCons 3 x)` where the syntactic sugar `new` is defined so that expressions after `new` are guaranteed to get unique identifiers. Note that `new` is not a function, but during typing it can behave like a function with type `(Id -> a) -> a`.

`new` can not be freely used. I call new-propagation the restriction that if a function right-hand side contains new-expression then the function should be used as a new-expression too. This is indicated by a `new` before the definition of the function:

```
new f x = new MyCons x (new MyNil)
```

Some usage of `f`:

```
      new f 1 === new f 1                      ⇓  False

        new x === new x    where x = f 1       ⇓  False

            y === y        where y = new f 1   ⇓   True
```

If identifiers are handled exclusively by `new` and `===` then the unique identifiers property holds. How to force this usage? My solution is to hide the type `Id`. The type `Id` can be hidden by the synonym data type

```
:: Create a  :==  Id -> a
```

and by an additional syntactic sugar `:::` which is defined so that

```
::: MyList a = MyCons a (MyList a)
             | MyNil
```

will replace the program lines at the beginning of this subsection.

1.2 Outline of the Paper

In Sect. 2 a simple conventional functional language is defined. This language is similar to the core language behind Clean[2]. Section 3 define the syntactic sugar `new` and `:::`. Some good properties of the syntactic sugars are shown in Sect. 6. In Sect. 5 a graph rewrite system is defined for the extended language.

2 THE CORE LANGUAGE

The core language is a simple functional language similar to the language behind Clean[2]. A program consists of definitions. Definitions are described by the BNF-like grammar

		Definitions	
D	$=$	f `::` σ	function type definition
	$\mid$	$f\ x_1 \ldots x_n$ `=` e	function definition
	$\mid$	`::` $T\ a_1 \ldots a_n$ `=` $A_1 \mid \ldots \mid A_n$	algebraic type definition
A	$=$	$C\ \sigma_1 \ldots \sigma_n$	constructor definition
		Expressions	
e	$=$	x	variable
	$\mid$	C	constructor symbol
	$\mid$	f	function symbol
	$\mid$	$e\ e'$	application
	$\mid$	`let` x_1=e_1`;` $\ldots$ `;`x_n=e_n `in` e	let expression
	$\mid$	`case` e `of` L_1`;` $\ldots$ `;`L_n	case expression
L	$=$	$C\ x_1 \ldots x_n$ `->` e	case alternative
		Types	
σ	$=$	a	type variable
	$\mid$	$T\ \sigma_1 \ldots \sigma_n$	compound type
	$\mid$	σ `->` σ'	function type

Here n is a natural number $(0, 1, 2, \ldots)$; x, a, f are sequences over the English alphabet beginning with lowercase letter; C and T (type constructor) are sequences over the English alphabet beginning with uppercase letter.

Function definitions, algebraic type definitions and case alternatives have so called left-hand side and right-hand side. The arity function for constructors, type constructors, and functions, and the number of free variables, $\mathrm{FV}(e)$ and $\mathrm{FV}(\sigma)$ are defined as usual.

Parenthesis, semicolons, bars and so called margin rules will be used to distinguish different syntactical units.

Syntactic restrictions:

Grammar segment	Restrictions
$f\ x_1 \ldots x_n$ `=` e	$i \neq j \Rightarrow x_i \neq x_j$, $\mathrm{FV}(e) \subseteq \{x_1, \ldots, x_n\}$
`::` $T\ a_1 \ldots a_n$ `=` $A_1 \mid \ldots \mid A_n$	$i \neq j \Rightarrow a_i \neq a_j$, $\mathrm{FV}(A_i) \subseteq \{a_1, \ldots, a_n\}$
$C\ x_1 \ldots x_n$	$i \neq j \Rightarrow x_i \neq x_j$, $n = \mathrm{arity}(C)$
$T\ \sigma_1 \ldots \sigma_n$	$n = \mathrm{arity}(T)$

Other restrictions:

- All used symbol is defined exactly once.
- The `start` function is defined with arity 0.

I will use additional language constructs in definitions, namely function alternatives, patterns, wildcards, local functions, global constants, guards, type classes and synonym data types. `Bool` is defined as `:: Bool = True | False`. In examples I will use well-known list notations too. I will use the Barendregt variable convention; i.e., if terms occur in certain mathematical context, then in these terms all bound variables are chosen to be different from the free variables [5].

Any conventional typing, both type-checking and type-inference, suit during the paper if it supports abstract types, synonym types and type classes.

The semantics can be defined either via lambda calculus or via a graph rewrite system. A graph rewriting semantics is used from Sect. 5. I will use the notation

$e \Downarrow \bot \quad \rightleftharpoons$ the head normal form of e is $\bot$ or the reduction do not stop,
$e \Downarrow e' \quad \rightleftharpoons$ the head normal form of e is e' ($e' \neq \bot$).

3 DEFINITION OF THE SYNTACTIC SUGAR

3.1 Syntax

Definition 3. *Extended programs* are programs with the syntax of the core language extended with the rules

D	$=$	`new` $f\ x_1 \ldots x_n$ `=` e	new-function
	$\mid$	`:::` $T\ a_1 \ldots a_n$ `=` $A_1 \mid \ldots \mid A_n$	`:::`-type definition
e	$=$	`new` e	new-expression

and with the predefined symbols

- `Create` is a type constructor symbol with arity 1.
- `===` is a function symbol with arity 2. `===` is used with infix notation.

Extended programs should obey the syntactic restrictions of the core language and the new-propagation rule.

Definition 4. *New-propagation* is the syntactic restriction that new-expressions are allowed only in new-functions and in the `start` function.

The new-propagation restriction is relaxed in Sect. 7.

Constructors of `:::`-types are called `:::`-constructors.

3.2 Semantics and Typing

Definition 5. Let Prelude be the following set of definitions:

```
:: Id = First | Left Id | Right Id

equal :: Id -> Id -> Bool
equal  First     First    = True
equal (Left  x) (Left  y) = equal x y
equal (Right x) (Right y) = equal x y
equal  _         _        = False

class getId :: a -> Id

(===) :: a -> b -> Bool   | getId a & getId b
(===) a b = equal (getId a) (getId b)

:: Create a :== Id -> a
```

Definition 6. The type transformation function τ on extended programs is defined by the rule

$$\texttt{:::}\ T\ a_1 \ldots a_n\ \texttt{=}\ C_1\ \sigma_{11} \ldots \sigma_{1m_1} \mid \ldots \mid C_\ell\ \sigma_{\ell 1} \ldots \sigma_{\ell m_\ell}$$
$$\longrightarrow$$
$$\begin{aligned}
&\texttt{::}\ T\ a_1 \ldots a_n\ \texttt{=}\ C_1\ \sigma_{11} \ldots \sigma_{1m_1}\ \texttt{Id} \mid \ldots \mid C_\ell\ \sigma_{\ell 1} \ldots \sigma_{\ell m_\ell}\ \texttt{Id;}\\
&\texttt{instance getId}\ T\ a_1 \ldots a_n\ \ \texttt{where}\\
&\qquad \texttt{getId}\ (C_1\ _\ \ldots\ _\ \texttt{i}) = \texttt{i}\\
&\qquad \ldots\\
&\qquad \texttt{getId}\ (C_\ell\ _\ \ldots\ _\ \texttt{i}) = \texttt{i}
\end{aligned}$$

Definition 7. Let P be an extended program. $\Phi(P) = \varphi(\tau(P)) \cup \text{Prelude}$, where φ is defined by the rules

1. $$\texttt{start} = e \quad\longrightarrow\quad \texttt{start} = \texttt{let id} = \texttt{First in}\ \psi(e)$$
2. $$\texttt{new}\ f\ x_1 \ldots x_n = e \quad\longrightarrow\quad f\ x_1 \ldots x_n\ \texttt{id} = \psi(e)$$

where ψ is defined by the rules

1. In case alternative left-hand sides, if C is a `:::`-constructor
$$C\ x_1 \ldots x_n \quad\longrightarrow\quad C\ x_1 \ldots x_n\ \_$$
2. $$\texttt{new}\ e \quad\longrightarrow\quad e\ \xi$$

where

$$\xi = \eta_1(\ldots(\eta_n\ \texttt{id})), \quad \eta_1,\ldots,\eta_n \in \{\texttt{Left},\texttt{Right}\},\ n \in \{1,2,3,\ldots\}$$

is chosen so that for each application of ψ the used ξ's are syntactically different.

The function ψ is not uniquely defined. To settle this fact, suppose that the ξ's are chosen systematically. Assume that the identifiers `id`, `First`, `Left`, `Right`, `equal` and `getId` is not used in extended programs.

Example 8. $\psi($`(new Nil, new Nil, new Nil)`$) =$
`(Nil (Left id), Nil (Left(Right id)), Nil (Right(Right id)))`.

Remark 9. It is easy to see that for each extended program P, $\Psi(P)$ is a program in the core language.

Definition 10. The P extended program is typeable if $\Phi(P)$ is typeable.

Remark 11. Lemma 18 will provide simplified typing rules.

Definition 12. e is a `:::`-expression if it has type $T\ \sigma_1 \ldots \sigma_n$ where T is a `:::`-type.

Remark 13. e is `:::`-expression iff there is a `getId` instance for the type of e.

3.3 Integrating Additional Language Constructs to the Extended Language

To show the viability the extended language I show the integration of three well-know language constructs into the extended language. This integration is not vital in the theory but I will need them in the examples.

Patterns can be easily integrated into the extended language: The first rule in the definition of ψ have to be generalised to patterns too.

To integrate function alternatives, no changes needed. Note from the typing rules follows that either all function alternative is a new-function alternative or none of them is a new-function alternative.

To integrate local functions the new-propagation rule have to be modified so that not only new-functions may contain local new-functions. For example, the following function definition is correct:

```
f x = g where new g y = new x
```

4 A COMPLETE EXAMPLE

A program in the extended language, which creates a ring from a list and then computes the length of the ring:

```
::: Ring a = Cons a (Ring a)

ringFromList :: [a] -> Create (Ring a)
new ringFromList l = r
where
    r = new f r l

    new f end []    = end
    new f end [h:t] = new Cons h (new f end t)

next :: Ring a -> Ring a
next (Cons _ x) = x

ringLength :: Ring a -> Int
ringLength a = f a (next a)
where
  f a b
     | a == b    = 1
     | otherwise = 1 + f a (next b)

start = ringLength (new ringFromList [1,2,3,1,2,3])
```

The program transformed by Φ, without the Prelude:

```
:: Ring a = Cons a (Ring a) Id

instance getId (Ring a) where
    getId (Cons _ _ id) = id

ringFromList :: [a] -> Id -> Ring a
ringFromList l id = r
where
    r = f r l id

    f end []    id = end
    f end [h:t] id = Cons h (f end t (Left id)) (Right id)

next :: (Ring a) -> Ring a
next (Cons _ x _) = x

ringLength :: (Ring a) -> Int
ringLength a = f a (next a)
where
   f a b
     | equal (getId a) (getId b) = 1
     | otherwise                 = 1 + f a (next b)

start = let
    id = First
        in ringLength (ringFromList [1,2,3,1,2,3] id)
```

5 THE EXTENDED GRAPH REWRITE SYSTEM

5.1 Graph Rewrite System Summary

Graph rewriting is a suitable technique to implement lazy functional languages efficiently. A computation in a graph rewrite system is specified by a set of graph rewrite rules that are used to rewrite a given initial graph to its final result. The graph rewrite system defined here is the same as the graph rewrite system behind the Clean functional language [6].

The graphs used during the rewrite process are called *data graphs*. Data graphs are directed and connected graphs with a signed root. Each node contains a *symbol* which can be either a *constructor symbol* or a *function symbol*. A node with symbol *S* will be called *S*-node. There are two special nodes, the *empty node* which has no outgoing edges and the *application node* which has two outgoing edges. Each symbol have a fixed *arity*, a natural number. There is a map from functional expression to data graphs defined in [1].

Rewrite rules specify transformations of data graphs. A rewrite rule is similar to a data graph but it has two signed root and may contain *variable nodes*. The two part of a rewrite rule which can be reached from the two root is called *left-hand side* and *right-hand side*.

A rewrite rule *matches* a subgraph of a data graph if there is a graph homeomorphism from the left-hand side of the rule to the subgraph. (Homeomorphism between data graphs should preserve symbols of nodes.) The matched subgraph is called *redex* (reducible expression). If a rule matches a redex, the data graph can be rewritten according to the rule. First the right-hand side is added to the data graph as separate nodes (variable nodes are treated specially) then all edges pointing to the image of root of the left-hand side are redirected to image of the root of the right-hand side. Finally in the *garbage collection* phase the nodes which are not reachable from the root of the data graph are removed.

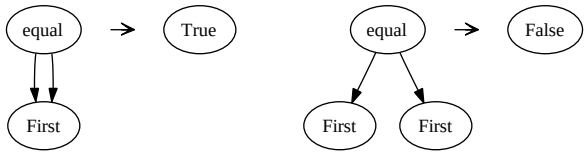
A *graph rewrite system* is a collection of rewrite rules. In a *priority graph rewrite system* rewrite rules are ordered. There is a map from functional programs to priority rewrite systems. [1]. This mapping first transforms case expressions into function alternatives. The ordering of the rewrite rules are based on the ordering of the function alternatives. Each priority rewrite system which was generated from a functional program has a special form, for example left-hand sides of the rules are trees. These rewrite systems are called *functional rewrite systems*. Functional rewrite systems are *confluent*: for any two reduct of a data graph there is a common reduct of them.

A *reduction sequence* is sequence of graphs so that each graph is obtained by a reduction step from the previous one. A *reduction strategy* is a function giving a particular reduction sequence of a data graph so that if the sequence has a last graph then it can not be reduced further. I will use functional reduction strategy [1].

5.2 The Modified Graph Rewrite System

Definition 14. Let the *comparing rewrite steps* be the following steps: Any `equal`-node with two out edges can be replaced by a `True`-node if the two out edges point to the `First`-node with no out edges. Any `equal`-node with two out edges can be replaced by a `False`-node if the two out edges point to two different `First`-node with no out edges.

Remark 15. The comparing steps may be pictured as:



Definition 16. Let Prelude' be the following set of definitions:

```
:: Id = First
equal :: Id -> Id -> Bool
class getId :: a -> Id

(===) :: a -> b -> Bool   | getId a & getId b
(===) a b = equal (getId a) (getId b)

:: Create a :== Id -> a

new :: (Create a) -> a
new x = x First
```

Definition 17. Let P be an extended program. $\Phi'(P) = \varphi'(\tau(P)) \cup$ Prelude', where φ' is defined by the rules

1. For each new-function left-hand side

$$\texttt{new}\ f\ x_1 \ldots x_n \quad \longrightarrow \quad f\ x_1 \ldots x_n \_$$

2. For each `:::`-constructor in patterns and case expressions

$$C\ x_1 \ldots x_n \quad \longrightarrow \quad C\ x_1 \ldots x_n \_$$

Lemma 18. $\Phi(P)$ is typeable iff $\Phi'(P)$ is typeable and the transformed new-functions has return type `Create` σ for some type σ.

Theorem 19. Let P be a typeable extended program where `start` has type `Bool`. The graph rewrite system described by $\Phi(P)$ and the graph rewrite system described by $\Phi'(P)$ with the comparing rewrite steps reduce `start` to the same expression, i.e. `start` reduces either to `True`, `False` or $\bot$, or the reduction will not stop in both cases.

Proof. A) Suppose that the reduction sequence of the `start` expression of $\Phi(P)$ with the functional reduction startegy is the finite reduction sequence $G_1 \to G_2 \to \ldots \to G_n$ where G_n is either a `True`-node, a `False`-node or a $\bot$-node.

In G_i ($i = 1, \ldots, n$) first replace all `Left` and `Right`-nodes by `First`-nodes with no out edges, then collect all unused nodes. In the resulted graph sequence erase all repeating occurrences of the same graph. We get a sequence of graphs G'_1, G'_2, G'_m where $G'_1 = G_1$, $G'_m = G_n$ and $m \leq n$. $G'_1 \to G'_2 \to \ldots \to G'_m$ is a reduction sequence of of $\Phi'(P)$. The same process can be done if the reduction sequence of $\Phi(P)$ is infinite.

B) Let $G_1 \to G_2 \to \ldots \to G_n$ be any finite reduction sequence of $\Phi(P)$ where the graph G_1 contains the `start`-node. By suitable replacement of `First`-nodes by chains of `Left`-nodes and `Right`-nodes, and by replacing comparing rewrite steps by sequences of rewrite steps of `equal`, we can reach a reduction sequence of $\Phi(P)$. The similar is true if the reduction sequence of $\Phi(P)$ was infinite.

If we put A) and B) together the theorem is proved.

Remark 20. The implementation cost of `new` and `===` is small. Even this small cost can be lowered by a more sophisticated graph rewrite system: `new` and `getId` could be totally eliminated, and the rewrite rule of `===` could be modified so that it reduces to `False` if its arguments are different graph nodes in root normal form and it reduces to `True` if its arguments are the same graph node.

6 PROPERTIES OF THE EXTENDED LANGUAGE

Note that theorems in this section can be stated with any semantics of `new`. They are placed after the definition of graph rewriting semantics, because some proof will use that semantics.

Suppose that P is a fixed extended program.

Theorem 21. Let e, e' and e'' be `:::`-expressions.

$$e \texttt{ === } e' \Downarrow \bot \quad \Leftrightarrow \quad e \Downarrow \bot \vee e' \Downarrow \bot$$

$$e \not\Downarrow \bot \quad \Rightarrow \quad \texttt{let } x = e \texttt{ in } x \texttt{ === } x \Downarrow \texttt{True}$$

$$e \texttt{ === } e' \Downarrow \texttt{True} \quad \Leftrightarrow \quad e' \texttt{ === } e \Downarrow \texttt{True}$$

$$e \texttt{ === } e' \Downarrow \texttt{True} \wedge e' \texttt{ === } e'' \Downarrow \texttt{True} \quad \Rightarrow \quad e \texttt{ === } e'' \Downarrow \texttt{True}$$

Proof. Note that a `===` b reduces to `equal (getId` a`) (getId` b`)`. The right-to-left inference in the first statement follows from that `equal` and all instances of `getId` are strict in their arguments. The left-to-right inference in the first statement follows from that $\bot$ of type `Id` can not be created in any program, and that `equal` and all instances of `getId` are total functions. The other statements follows from the definition of `equal` (for example, `equal` is a reflexive and symmetric function).

Definition 22. e and e' are contextually equivalent[3], denoted by $e \approx e'$, iff for each function f, if $f\ e$ typeable with type `Bool` then $f\ e'$ is typeable with type `Bool` and $f\ e$ and $f\ e'$ have the same reduct (either `True`, `False` or $\perp$).

Theorem 23. Let e and e' be `:::`-expressions.

$$e \texttt{ === } e' \Downarrow \texttt{True} \quad \Rightarrow \quad e \approx e'.$$

Proof. `equal (getId` e`) (getId` e'`)` $\Downarrow$ `True` is possible only if the head normal reduct of the two `getId` is the same node. This is possible only if the root of e and e' is the same node. From that follows that e and e' share all properties.

Theorem 24. Let `new` e `:::`-expression.

$$\texttt{new}\ e \not\Downarrow \perp \quad \Rightarrow \quad \texttt{new}\ e \texttt{ === new } e \Downarrow \texttt{False}$$

Proof. `new` e `===` `new` e is equivalent to $e\ \xi$`===` $e\ \xi'$ where ξ and ξ' are syntactically different expressions created by applications of `Left` and `Right` on `id`. $e\ \xi$`===` $e\ \xi'$ reduces to `equal` $\xi\ \xi'$ which reduces to `False`.

Conjecture 25. If e and e' are expressions so that their graphs G_1 and G_2 differs only in copied nodes (there is a graph G so that G homeomorph to G_1 and G_2) then $e \approx e'$.

Theorem 26.

$$\texttt{let } x \texttt{ = } e \texttt{ in } e' \quad \approx \quad e'[e/x].$$

holds for each expression e and e' if x occurs at most once in e' or e does not contain new-expression.

Proof. If x occurs at most once in e' than `let` x `=` e `in` e' and $e'[e/x]$ have the same graph. If e does not contain new-expression then `let` x `=` e `in` e' and $e'[e/x]$ differs only by copied graph nodes.

Conjecture 27. For each definition `new` $f\ x_1 \ldots x_n$ `=` e and each expression e

$$\texttt{new}\ f\ e_1 \ldots e_n \quad \approx \quad e[e_1/x_1, \ldots, e_n/x_n].$$

Remark 28. The previous theorems and conjectures suggest a structural operational semantics of `new`.

7 RELAXING THE NEW-PROPAGATION RESTRICTION

Definition 29. An algebraic type constructor is an identifier-free type constructor if it is not a `:::`-type constructor and its definition contains only identifier-free types. A type is an identifier-free type if it contains only type variables, identifier-free type constructors and function types.

Example 30. Identifier-free types are exactly those which can be defined without the `:::` syntactic sugar. Some of these: `Bool`, `List Bool`, `List a`, `(List a) -> Bool`.

Definition 31. The *relaxed new-propagation rule* says that only new-functions and functions with identifier-free types may contain new-expressions.

Conjecture 32. All the previous theorems hold with the relaxed new-propagation rule and with the modified version of the first rule of φ (φ is defined in Def. 7):

1' If f has identifier-free type and e contains new-expression

$$\texttt{f}\ x_1 \ldots x_n = e \quad \longrightarrow \quad \texttt{f}\ x_1 \ldots x_n = \texttt{let id = First in}\ \psi(e)$$

Example 33. The following function is now accepted:

```
len x = ringLength (new ringFromList x)
```

8 PROTOTYPE IMPLEMENTATION IN CLEAN

I have a prototype implementation in Clean. The only non-functional part, the comparing rewrite steps are implemented in ABC language, which is the intermediate language of Clean. It is allowed to inline ABC code in Clean programs, so the prototype implementation is placed in a normal Clean module.

The ABC machine is a stack machine with 3 stacks, it does not have registers. The graph rewriting is performed in a heap (or graph store) in which nodes can be created and overwritten. The *A-stack* stores pointers to nodes, the *B-stack* stores non-pointers, like integers, booleans, characters and floating point values, and the *C-stack* pointers to code. The A and C-stack are separated to make garbage collection easier. The B-stack is used to make computations using integers, booleans characters and floating point numbers faster.

Any Clean module (except the main module) consist of two files, the *definition module* and the *implementation module*. The definition module of the prototype implementation:

```
definition module eq
import StdEnv

:: Id
:: Create a :== Id -> a
new :: (Create a) -> a
class getId a :: a -> Id
(===) :: a b -> Bool | getId a & getId b
```

The implementation module:

```
implementation module eq
```

```
:: Id = Wrap Id

new :: (Create a) -> a
new x = x i where i = Wrap i

(===) :: a b -> Bool | getId a & getId b
(===) a b = equal (getId a) (getId b)

equal (Wrap x) (Wrap y) = compare_pointer x y

compare_pointer :: !a !b -> Bool
compare_pointer a b = code {
    push_a_b 0
    pop_a 1
    push_a_b 0
    pop_a 1
    eqI
}
```

Instead of the type `:: Id = First` and the identifier value `First` I use the type `:: Id = Wrap Id` and the value `let i = Wrap i in i` because the Clean compiler optimise out heap allocations for constructors without arguments. The implementation of `equal` eliminates one `Wrap` from the identifier values because identifier values could be unravelled by one level when a closure is overwritten by them. Please refer to [7].

`compare_pointer` has two strict arguments. (The demand for strictness is denoted by an exclamation mark `!` before the type of the argument.) A the moment that the control is transferred to the ABC code above the arguments are evaluated in the heap and its pointers are in the top of the A-stack. The `push_a_b 0` instruction pushes the top element of the A-stack into the B-stack. The `pop_a 1` instruction pops the top element of the A-stack. After the first four ABC instructions the pointers of the two arguments will be on the top of the B-stack. The `eqI` instruction pops the two top element of the B stack and compares them as integers. (Clean integers have the same size as pointers.) The result is stored on the B stack. That is the last step because functions with return types `Bool` should leave the return value on the B stack. (Thanks to John van Groningen for the details and for the ABC code segment.)

The two limitations of the prototype implementation:

- Left-hand side of new-functions and patterns of `:::`-constructors should be written in the transformed form defined by φ'. (φ' is defined in Def. 17.)
- New-propagation is not checked.

The ring example in the prototype implementation:

```
module ring_example
import StdEnv, eq
```

```
:: Ring a = MyCons a (Ring a) Id

instance getId (Ring a)  where getId (MyCons _ _ i) = i

ringFromList :: [a] Id -> Ring a
ringFromList l _ = r
where
    r = new (f l)

    f []    _ = r
    f [h:t] _ = new (MyCons h (new (f t)))

next :: (Ring a) -> Ring a
next (MyCons _ x _) = x

ringLength :: (Ring a) -> Int
ringLength a = f a (next a)
where
    f a b
        | a === b   = 1
        | otherwise = 1 + f a (next b)

Start = ringLength (new (ringFromList [1,2,3,1,2,3]))
```

9 RELATED WORKS

9.1 Relation with the ν-Calculus

The standard form of ν-calculus [3]:

$$
\begin{array}{rcll}
M & = & x & \text{variable,} \\
 & | & n & \text{name,} \\
 & | & \mathit{true} \mid \mathit{false} & \text{truth values,} \\
 & | & \mathit{if}\ M\ \mathit{then}\ M\ \mathit{else}\ M & \text{conditional,} \\
 & | & M = M & \text{compare names,} \\
 & | & \nu n.M & \text{create new name,} \\
 & | & \lambda x : \sigma.M & \text{function abstraction,} \\
 & | & M\ M & \text{function application.}
\end{array}
$$

If we enrich the ν-calculus with function definitions, case expressions, and let expressions, the `new` *e* syntactic sugar can be expressed as $\nu n.(e\ n)$.

There are two main differences between the ν-calculus and `new`. Firstly, ν calculus does not have a controlled way to bind names to expressions, so it is not possible to express and fulfil the unique identifiers property. Secondly, there is nothing similar to new-propagation in ν-calculus, so it has derivations like $x(\lambda z.z) = x(\lambda z.z) \Downarrow \mathit{false}$ if x is defined as $\lambda y.\nu n.n$, which means that there is an expression

which does not equal to itself. In the proposed language, e `===` $e \Downarrow$ `False` is possible only if e contains at least one `new` syntactically (see Theorem 26).

9.2 Future Works

This paper is part of a work in which pointer assignment would be modelled in pure functional languages. The soundness of this language element would be guaranteed by a mechanism similar to uniqueness typing.

These new language elements may also be a ground for defining abstract objects in functional languages [4].

10 CONCLUSION

I defined a syntactic sugar called `new` in a pure functional language. `new` provides easy-to-use unique identifiers. I described a graph rewriting system which is ground of an efficient implementation of `new`. I gave a prototype implementation of this rewrite system with the help of the Clean compiler. I showed several properties of `new` by which a structural operational semantics of `new` could be described.

REFERENCES

[1] M.J. Plasmeijer and M.C.J.D van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[2] Rinus Plasmeijer and Marko van Eekelen. Language report concurrent clean, 1998.

[3] Ian Stark. *Names and Higher-Order Functions*. PhD thesis, 1994.

[4] Máté Tejfel, Zoltán Horváth, and Tamás Kozsik. Extending the sparkle core language with object abstraction. *Acta Cybernetica*, 17:419–445, 2005.

[5] Christian Urban and Michael Norrish. A formal treatment of the barendregt variable convention in rule inductions. *Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding, Tallinn, Estonia*, pages 25–32, 2005.

[6] Marko C. J. D. van Eekelen, Sjaak Smetsers, and Marinus J. Plasmeijer. Graph rewriting semantics for functional programming languages. In *CSL '96: Selected Papers from the10th International Workshop on Computer Science Logic*, pages 106–128, London, UK, 1997. Springer-Verlag.

[7] J.H.G van Groningen. Implementing the abc-machine on m680x0 based architectures. Technical report, Department of Computer Science, University of Nijmegen, 1991.