A Simple Categorical Model of Reference Type

Novitzká, Valerie; Steingartner, William and Perháč, Ján

Abstract: ***Simply-typed λ-calculus is a well-known part of the type systems frequently used in real programming languages. It is described and modeled in many publications. In this paper, we extend this calculus with reference type that models a special part of a computer memory called dynamic memory. Reference types allows to use computer memory more flexibly, because we reserve only as much memory cells as are needed for computations. We shortly explain a fragment of simply-typed λ-calculus and its model as a cartesian closed category. Then we introduce the reference type together with its operations. We discuss the possibilities, how to model reference type in a category of types. The simplicity of the model and its graphical output make our approach useful for educational purposes.***

Index Terms: ***type theory, reference type, category theory, λ-calculus***

1. Introduction

Types form the essential part of many programming languages. Well-designed type system and sophisticated type-checking in a compiler can avoid many undesirable program errors. The formal basis of type systems offers type theory [9], [14]. From its beginning ideas formulated by Russell and later explicitly defined by Church as λ-calculus, this theory has been developed depending on the progress in computer science. Type theory was later extended by popular, not simple types as reference types, polymorphic types, exception types, subtypes, and dependent types [7].

In this paper, we construct an appropriate and simple model of reference type as an extension of simply-typed λ-calculus. Reference types use dynamic memory, heap, instead of stack memory. Dynamic memory is separately managed using addresses of stored values and it allows to use computer memory more effectively. We construct our model based on category theory [6]. Categories are useful mathematical structures for modeling programs written in various programming paradigms. The graphical illustration of categories also contributes to easier understanding of the models.

Our model is quite simple but we tried to save its exactness. It is dedicated for educational purpose, namely in the course Type Theory for graduate students. Therefore we give an impact on the principles of categorical models and we abstract from some technical details. We have used successfuly the appropriate categorical models in the course Semantics of Programming Languages, where the denotational semantics is modeled as a collection of categories of states [17] and the operational semantics as a coalgebra [16].

The structure of this paper is as follows. In the next section, we shortly characterise a fragment of simply-typed λ-calculus and its categorical model, cartesian closed category (*ccc*) that we use as a basis for modeling reference type. In Section 3, we introduce the basic concepts for reference types and we characterize the operations on them. In Section 4, we extend the categorical model of simple types by a new object for dynamic memory. We define the morphisms that represent operations on reference types.

2. Related Works

There are many publications dealing with reference types in various programming languages, e.g. for Java [10], C# [18], Python [1] and Swift [2]. They are oriented mainly on the details, how to use references in programming, and what is the difference between value types and reference types.

Naumov presents in [12] simply-typed λ-calculus extended with reference types and he defines their principles, syntax and semantics. Some other nice publications introducing a formal definition of reference types in the frame of λ-calculus are the books written by Pierce [13]–[15], which precisely describe motivation, formal definition, models of reference types in functional languages, and also their usage. An introduction into formalization of type theory is in [9] and in many other publications, e.g. in [3], [5], [19].

The categorical modeling of simple λ-calculus is published in the book of Crole [8], but it does not deal with reference types.

Manuscript received June, 2020.
This work has been supported by the project KEGA 011TUKE-4/2020: "A development of the new semantic technologies in educating of young IT experts".

Novitzká, Valerie. Author (contact person) works at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: valerie.novitzka@tuke.sk).

Steingartner, William. Author works at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: william.steingartner@tuke.sk)

Perháč, Ján. Author works at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: jan.perhac@tuke.sk).

3. Categorical Model of Lambda-Calculus with Function Types

Simply-typed λ-calculus is a basis for the functional programming languages [11]. Its categorical model is a cartesian closed category [8]. In this section we shortly repeat the principles of this model.

We consider a fragment of simply-typed λ-calculus, that contains basic types and function types. The formal syntax of the language consists of the pure λ-calculus extended with two terms of successor and a conditional term:

$$
\begin{array}{ll}
t ::= & x \mid \lambda x : T.t \mid t\ t \mid \texttt{unit} \mid \texttt{succ}\ t \\
 & \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\
v ::= & nv \mid \texttt{true} \mid \texttt{false} \mid \lambda x : T.t \mid \texttt{unit} \\
nv ::= & 0 \mid \texttt{succ}\ nv \\
T ::= & \texttt{Nat} \mid \texttt{Bool} \mid T \to T
\end{array}
$$

We consider the basic types Nat for natural numbers, $Bool$ for boolean values and $Unit$ as an empty type. The only simple type is the function type $T \to T$. We do not consider other simple types, as product types and sum (coproduct) types, but the principles of modeling that language are the same.

A program is a term t. A term can be a variable x, a λ-abstraction of the form $\lambda x : T.t$ defining a function, an application $t\ t$, a unit (empty) term $\texttt{unit}$, a successor $\texttt{succ}\ t$ and a conditional term $\texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t$. Besides numerical values (nv) and truth values there are also λ-abstractions and an empty value among the values.

For constructing a categorical model of this language, we need to choose a suitable kind of a category. A category is a mathematical structure consisting of

- objects, and
- morphisms between objects,

and it has to satisfy the following basic axioms:

- each object has the identity morphism;
- for two composable morphisms, there exists a morphism that is their composition;
- a composition of the composable morphisms is associative.

A cartesian closed category (*ccc*) is the best and traditional solution for defining a model of this language. A *ccc* has the following properties:

- it has a terminal object 1, i.e. an object to which there exists just one morphism from each category object,
- each pair of objects A and B has a product object $A \times B$ with two projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$,
- for every pair of objects A and B there exists an exponent object B^A representing the functions from A to B, and a morphism

$$eval : B^A \times A \to B.$$

The constants, the elements of an object can be defined as global variables, the morphisms from terminal object to a given object. The precise definition of cartesian closed categories is published in [4], [6] and in many others.

To build a model of our language, we need first to define the representations of the types. We assign to the types from the syntax the following representations:

$$
\begin{array}{l}
\texttt{Nat} \mapsto \mathbf{N}_0, \\
\texttt{Bool} \mapsto \mathbf{B}, \\
T_1 \to T_2 \mapsto [\![T_2]\!]^{[\![T_1]\!]}, \\
\texttt{Unit} \mapsto 1,
\end{array}
$$

where $\mathbf{N}_0$ is a set of natural number with zero and $\mathbf{B}$ is a set of truth values.

The model of simply-typed λ-calculus is the category $\mathbf{C}$ defined as follows:

- the objects are the representations of types,
- the morphisms are the representations of terms,
- the terminal object 1 is the representation of the type $\texttt{Unit}$, $[\![\texttt{Unit}]\!] = 1$.

A term $x : T$ is represented as the identity morphism on $[\![T]\!]$:

$$id_{[\![T]\!]} = [\![x]\!].$$

A term $\lambda x : T_1.t$ is a function definition of a function type $T_1 \to T_2$, where $t : T_2$. A λ-abstraction is a value of a type representation $[\![T_2]\!]^{[\![T_1]\!]}$. Values are modeled as the morphisms (global variables) from terminal object:

$$[\![\lambda x : T_1.t]\!] : 1 \to [\![T_2]\!]^{[\![T_1]\!]}.$$

An application $t_1\ t_2$, where $t_1 : T_1 \to T_2$ is a function and $t_2 : T_1$ is its argument, is represented as a morphism

$$[\![t_1\ t_2]\!] : [\![T_2]\!]^{[\![T_1]\!]} \times [\![T_1]\!] \to [\![T_2]\!]$$

and defined as a composition:

$$[\![t_1\ t_2]\!] = [\![t_2]\!] \circ [\![t_1]\!].$$

The constants $\texttt{true}, \texttt{false} : \texttt{Bool}$ are represented as the morphisms (global variables) from terminal object to $\mathbf{B}$:

$$[\![\texttt{true}]\!], [\![\texttt{false}]\!] : 1 \to \mathbf{B}$$

and the constant 0 of the type $\mathbf{N}_0$ is the morphism:

$$zero : 1 \to \mathbf{N}_0.$$

Similarly, the representation of a term $\texttt{succ}\ t : \texttt{Nat}$ is an endomorphism on natural numbers $\mathbf{N}_0$

$$[\![\texttt{succ}\ t]\!] : \mathbf{N}_0 \to \mathbf{N}_0,$$

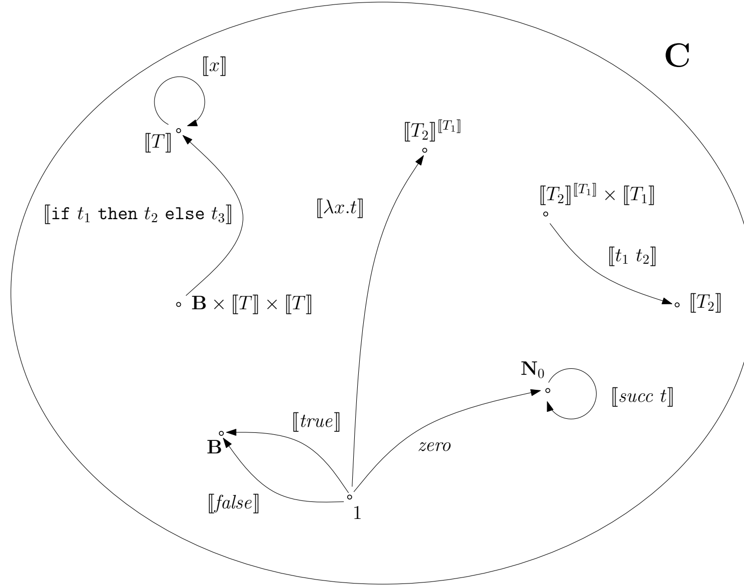

Fig. 1: A model of simply-typed λ-calculus

and the representation of a conditional term $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$ is a morphism

$$[\![\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3]\!] : \mathbf{B} \times [\![T]\!] \times [\![T]\!] \to [\![T]\!],$$

where $t_1 : \texttt{Bool}$ and $t_2, t_3 : T$.

The advantage of categorical model is its graphical representation as we illustrate it in Fig 1.

4. Reference Type

Most programming languages use references that are often called pointers. They enable to work explicitly with some part of memory. A reference is a value that defines an access to a memory cell, i.e. an address, where some value is stored. The work with references is usually called indirect addressing. The part of the memory used by references is called dynamic memory, or heap. References increase flexibility of memory usage, because we reserve only so much memory that we explicitly need. Typical application of references are the linked lists.

A collection of references together with their operations form the reference type, $\texttt{Ref}\ T$. To avoid confusion, a reserved place has to be of some type T. To work with references, the following operations are needed:

- reservation, or allocation of a place (location) in dynamic memory to store a value of a given type, $\texttt{alloc}\ t$;
- dereference, i.e. extraction of a value stored in a given place in dynamic memory, $@t$;
- assignment, i.e. modification of a value stored in a given place in dynamic memory, $t_1 := t_2$.

We extend the syntax of our fragment of λ-calculus with the following structures:

$$\begin{array}{ll} t ::= & \ldots \mid \texttt{alloc}\ t \mid @t \mid t := t \mid l \\ v ::= & \ldots \mid l \\ T ::= & \ldots \mid \texttt{Ref}\ T \end{array}$$

The operation $\texttt{alloc}$ allocates a new memory cell and it stores there a value of a term t. A type of this operation is:

$$\texttt{alloc}\ t : \texttt{Ref}\ T \quad \text{for} \quad t : T.$$

To evaluate this term, first, a term t has to be reduced to a value (if it is a redex)

$$t \to^* v$$

and this value is stored in reserved place at the first free location. The symbol $\to^*$ denotes finite number of reduction steps.

The operation $@$ is dereference; it returns a value stored in a location t. The type of this operation is:

$$@t : T \quad \text{if} \quad t : \texttt{Ref}\ T.$$

Again, if t is a redex, it has to be reduced to a location:

$$t \to^* l$$

and $@l$ returns a value stored at l.

The operation $:=$ is assignment. It serves for modification of a value stored in dynamic memory. Therefore, the evaluation of

$$t_1 := t_2$$

means that $t_1 : \texttt{Ref}\ T$ is reduced to a location:

$$t_1 \to^* \ l,$$

then $t_2 : T$ is reduced to a value:

$$t_2 \to^* \ v$$

and v is stored at the location l. The assignment is made indirectly, therefore, the type of this operation is `Unit`.

The arithmetic on locations are obviously not allowed in programming languages, because it can make confusion in address space. For the same reasons, the release of allocated cells has no operation in programming languages, it is mainly done by garbage collector.

5. *Categorical Model of Reference Type*

To model our extended language from the previous sections with reference type, we need to modify the category $\mathbf{C}$ and to introduce a representation of reference type together with new morphisms for the operations on this type. That means, we need to model and to handle dynamic memory.

In modeling dynamic memory, we need to save an information about a type T for the values that can be stored in a reserved location. To model dynamic memory, we have two possibilities. The first is to model reference type for each type T separately. Such approach brings some technical complications, of how to handle locations of dynamic memory in different objects. Because this model is addressed mainly for the students at the technical university that have a little experience with category theory, we make our model the simplest possible. The main purpose is that students understand the principles without technical details.

Therefore, we model an abstraction of computer memory as a table of memory cells, which is indexed by locations and the values are stored in the cells. In this first consideration, we do not consider different representation of the types of values, every memory cell, a location, can contain a value of any type. This simplification enables us to represent dynamic memory as a function type `Ref`:

$$\texttt{Dynam} = \texttt{Loc} \to \texttt{Values},$$

where `Loc` is a finite ordered set of memory locations

$$\texttt{Loc} \subset \mathbf{N},$$

and `Values` is a set of possible values stored in the memory cells.

To store the type annotations of the values in dynamic memory, we store it together with the value. We extend the representation of dynamic memory `Dynam` as a function space:

$$\texttt{Dynam} = \texttt{Loc} \to (\texttt{Value} \times \texttt{Types}),$$

where `Types` is a finite set of all type representations of a program

$$\texttt{Types} = \{[\![T_1]\!], \dots, [\![T_n]\!]\}.$$

Now, we have an information about the types of values that can be stored in the given location and dynamic memory is represented as a one category object, ordered set of functions. We can define an actual dynamic memory in any step of computation as an ordered set:

$$\{l_1 \mapsto ([\![v_1]\!], [\![T_1]\!]), \dots, l_i \mapsto (\bot, 1)\},$$

where the last member indicates a location of the first free memory cell with undefined value and type.

Now, we model the operations with reference type. an allocation of a new location is modeled as a morphism:

$$[\![\texttt{alloc}\ t]\!] : [\![T]\!] \to \texttt{Dynam},$$

defined for $t : T$ by:

$$\begin{aligned}[\![\texttt{alloc}]\!]\ [\![t]\!] = \quad & \{l_1 \mapsto ([\![v_1]\!], [\![T_1]\!]), \dots, \\ & l_i \mapsto ([\![t]\!], [\![T]\!]), l_{i+1} \mapsto (\bot, 1)\}.\end{aligned}$$

The operation of dereference is the only operation that goes out of a dynamic memory and it returns a value of a given type for further computation:

$$[\![@\ t]\!] : \texttt{Dynam} \to [\![T]\!]$$

and it is defined by

$$[\![@ l_j]\!] = [\![v_j]\!],$$

where the actual memory is

$$\begin{aligned}&\{l_1 \mapsto ([\![v_1]\!], [\![T_1]\!]), \dots, l_j \mapsto ([\![v_j]\!], [\![T]\!]), \dots, \\ &\quad l_i \mapsto (\bot, 1)\}.\end{aligned}$$

We model the assignment as an operation performed inside a dynamic memory:

$$[\![t_1 :=\ \ t_2]\!] : \texttt{Dynam} \to \texttt{Dynam}$$

and we define it for an actual memory

$$\begin{aligned}&\{l_1 \mapsto ([\![v_1]\!], [\![T_1]\!]), \dots, l_j \mapsto ([\![v_j]\!], [\![T_j]\!]), \dots, \\ &\quad l_i \mapsto (\bot, 1)\}.\end{aligned}$$

by

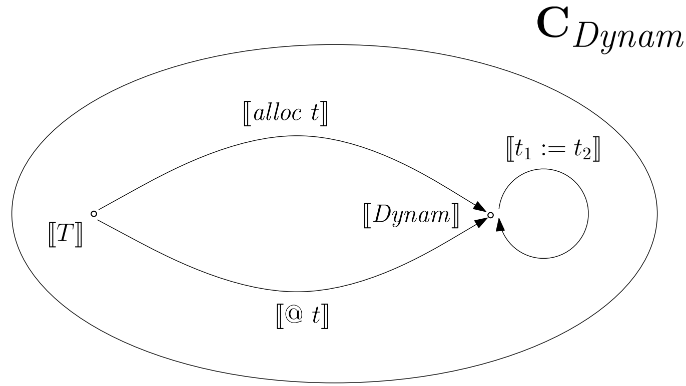

Fig. 2: Modeling dynamic memory

$$[\![l_j := \quad v]\!] = \{l_1 \mapsto ([\![v_1]\!], [\![T_1]\!]), \ldots,$$
$$l_j \mapsto ([\![v]\!], [\![T_j]\!]), \ldots, l_i \mapsto (\perp, 1)\}.$$

The operations on reference type are illustrated in Fig 2.

6. Conclusion

Using dynamic memory leads to economic and effective handling with computer memory. We extended a categorical model of simply-typed λ-calculus with a representation of dynamic memory. We tried to use existing model in the form of *ccc*. Our extensions with new object of dynamic memory as a special type together with appropriate morphisms preserve the simplicity and elegance of the previous model. Therefore, our approach is easily understandable by students and practical programmers that are not experts in theoretical computer science.

This result is the first step to introduce types in our categorical models of programming languages, especially for program systems consisting of components. We will use it and extend during our further research.

References

[1] Built-in types — python 3.8.2rc1 documentation. *docs.python.org*.
[2] Structures and classes — the swift programming language (swift 5.2). *docs.swift.org*.
[3] E. Ábrahám and M. Bonsangue. In *Theory and practice of formal methods*, 2016.
[4] S. Awodey. *Category theory*. Oxford University Press, New York, NY, USA, 1st edition, 2010.
[5] A.K. Bansal. *Introduction to programming languages*. CRC Press, New York, NY, USA, 1st edition, 2013.
[6] M. Barr and C. Wells. *Category theory of computing science*. Prentice Hall, New York, NY, USA, 1st edition, 1998.
[7] J. Collins. *A history of the theory of types*. Lambert Academic Publishing, New York, NY, USA, 1st edition, 2012.
[8] R. L. Crole. *Categories for Types*. Cambridge University Press, New York, NY, USA, 1st edition, 1993.
[9] Z. Csörnyei. *Introduction to the Type Theory (in hungarian)*. ELTE Eötvös Press, New York, NY, USA, 1st edition, 2012.
[10] P.J. Deitel and H. Deitel. *Introduction to Java 9 Classes, Objects, Methods and Strings*. Pearson, New York, NY, USA, 1st edition, 2017.
[11] W. Farmer. The seven virtues of simple type theory. *Journal of Applied Logic*, 6:267–286, 2008.
[12] P. Naumov. Theory of reference types. *Technical report, Cornell University*, 1998.
[13] B.C. Pierce. *Basic category theory for computer scientists*. MIT Press, New York, NY, USA, 1st edition, 1991.
[14] B.C. Pierce. *Types and programming languages*. MIT Press, New York, NY, USA, 1st edition, 2002.
[15] B.C. Pierce. *Advanced topics in types and programming languages*. MIT Press, New York, NY, USA, 1st edition, 2004.
[16] W. Steingartner, V. Novitzká, and W. Schreiner. A coalgebraic operational semantics. *Computing and Informatics*, 38:1181–1209, 2019.
[17] W. et al Steingartner. New approach to categorical semantics for procedural languages. *Computing and Informatics*, 36:1385–1414, 2017.
[18] R. Stephens. *$C\#$ 5.0 Programmer's Reference*. John Wiley et Sons, Indianapolis, 1st edition, 2014.
[19] S. Thompson. *Type theory and Functional programming*. University Kent, New York, NY, USA, 1st edition, 1999.

Valerie Novitzká works as Full Professor of Informatics at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. Her fields of research include type theory, semantics of programming languages, non-classical logical systems and their applications in computing science. She also works behavioural modeling of large program systems based on categories.

William Steingartner works as Assistant Professor of Informatics at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. He defended his Ph.D. thesis "The Rôle of Toposes in Informatics" in 2008. His main fields of research are semantics of programming languages, category theory, compilers, data structures and recursion theory. He also works with type theory and software engineering.

Ján Perháč works as Assistant Professor of Informatics at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. He defended his Ph.D. thesis "The Rôle of Toposes in Informatics" in 2019. His main fields of research are type theory and non-classical logical systems.