References in Pure Functional Languages

Summary of Ph.D. Thesis*

Péter Diviánszky

PhD School of Computer Science,
Eötvös Loránd University, Faculty of Informatics
PhD Program: Basics and methodology of informatics
Leader of the PhD School and Program:
Dr. András Benczúr

Department of Programming Languages and Compilers

Advisor: Dr. Zoltán Horváth

2012.

*The research was supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. T037742, by GVOP-3.2.2.-2004-07-0005/3.0 ELTE IKKK and OMAA-ÖAU 66öu2, by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003)

Introduction

Referential Transparency

Referential transparency plays an important role in the success of functional programming languages.

Given an environment, an expression and a variable, referential transparency holds if the variable can be replaced by its definition without changing the behaviour of the expression:

$\mathtt{e} \longrightarrow \mathtt{e}[\mathtt{v} := \mathtt{a}] \quad \text{if} \quad \{\ldots; \mathtt{v} = \mathtt{a}; \ldots\}$

Purely functional languages can be defined as languages in which referential transparency has no precondition, or referential transparency has a simple precondition. By this definition, Haskell, Clean[1] and Miranda are purely functional languages, for example.

The definition of referential transparency mentions behaviour of expressions and programs which should be explained. Here we mean functional behaviour, which excludes properties like time and memory consumption and includes properties like what output we will get given an input to the program.

Referential transparency is deeply connected to definiteness, unfoldability (β-reduction), substitutivity of identity (Leibniz's law), persistence, extensionality, determinacy and lack of side effects, whose relation is described in [14].

Advantages of referential transparency:

- easier program correctness proofs,
  the possibility to use equational reasoning,
- more code optimization steps,
  partial evaluation in compile time,
- more freedom to change the evaluation strategy in runtime,
  possibility of parallel evaluation.

Challenges with Referential Transparency

There are quite a few challenges to solve when referential transparency always holds. In this thesis I concentrate on the challenge of using references.

[1] In Clean, referential transparency is limited by uniqueness typing rules.

Solutions

I have collected the following methods for replacing references in purely functional languages:

1. Using purely functional data structures

   Summary References can be modelled by finite maps[1].

   Advantages Programmers gain additional freedom by using finite maps or similar data structures because purely functional data structures are persistent. It is possible to save and restore the referred values of all references at once, for example.

   Drawbacks Persistence makes destructive updates impossible which prevents the use of lots of efficient implementations. In particular, the cost of basic operations of finite maps is not constant, it increases by the number of elements in the map.

   In the thesis I defined a modified finite map data structure with constant cost of basic operations if some natural conditions hold. These conditions hold if we use finite maps for modelling references.

2. Tracking the side effect of computation steps

   If we manage to track the side effect of computation steps then we can benefit from the advantages of referential transparency in case of side-effect free computation steps at least.

   Methods to track side effects:

   (a) by proving program correctness

      Summary During program correctness proofs all information is available to track side effects. Separation logic[13] is suitable for tracking the side effects of reference operations for example.

      Advantages Program source code remains simple, because information about side effects are stored in the theorems and proofs about the program.

      Drawbacks Precondition of referential transparency is complex, and it is detached from the source code, thus tracking side effects by proving program correctness is not suitable for purely functional programming.

The information about side effects is hardly accessible during program compilation for optimization purposes.

If separation logic is available in the type system then the drawbacks of effect typing applies.

In the thesis I am not concerned with this method because I consider tracking side effects by proving program correctness not an option for purely functional programming.

(b) by effect typing

Summary Effect typing[17][12] tracks side effects by the type system.

Advantages Program source code remains simple, because information about side effects are stored in types and these types may be inferred automatically.

Drawbacks Precondition of referential transparency is complex which reduces the advantages of referential transparency. Equational reasoning is harder for example.

Effect type systems can grow quite complex, especially in case of lazy evaluation.

In the thesis In case of some side effects, like the side effect of reference comparison, the precondition of referential transparency can remain simple. In this case effect typing and similar methods are beneficial, which I show in the third thesis.

(c) by uniqueness typing

Summary Uniqueness typing[2] can be seen as simplified effect typing where the only possible effect is that a value was already used in an operation. Other side effects are expressed by this single side effect.

Advantages Referential transparency holds without precondition during reading the program code, for example during equational reasoning.

Program code remains simple, because information about side effects are stored in types which can be inferred automatically.

Drawbacks Precondition of referential transparency is complex during program writing, for example during program transformation steps.

The type system should be adapted to deal with uniqueness information (this can be done with uniqueness type attributes for example).

In the thesis In the second thesis I use a simplified version of uniqueness typing to track the side effects of reference operations.

(d) by monads

Summary Monads define how to compute the side effect of a compound expression from the side effects of its parts [11] [16].

Advantages Referential transparency holds without precondition.

Monads need no special typing rules, the type of monadic operations can be given by type constructors and universal polymorphism.

Drawbacks Using monads has an effect on source code, and sometimes different versions of functions are defined both for side effect free and effectful parameters.

Referential transparency holds without precondition but using monads may force the evaluation order which puts a restriction on the program correctness proof step orders and restricts optimization possibilities too, which are the symptoms of referential opacity.

Dealing with orthogonal side effects is not satisfactory with monads.

In the thesis I show the existing monadic models of references.

All four methods[2] has its own advantages and drawbacks, so we should chose the method which suits the best the given concrete application.

[2]Tracking side effects by proving program correctness is not an option for purely functional programming so four methods remain.

Theses

I show how references can be modelled in pure functional languages beyond the monadic solutions.

Thesis 1
References can be efficiently replaced by finite maps

I defined modified finite maps which are more suitable than finite maps as a replacement for references because of their improved efficiency.

Papers [6] and [7], related paper [8]

Finite maps are association lists in which key–value pairs are not ordered. References can be replaced by finite maps such that keys are used instead of references and the referred values are stored in a finite map.

If finite maps are used as a replacement for references then the following conditions usually hold:

- After changing the value of a key, the old value of the key is never read or changed (in the old map). This condition holds because it is not possible to refer to the runtime environment before the reference write.
- There is an upper limit for the number of finite map during runtime. There is only one, global environment for references, which is divided into finite maps during the replacement. The number of these finite maps are usually known in compile time.

Basic operations of modified finite maps, like write (insert), read (lookup), new key generation and key comparison have constant cost if the above two conditions hold. This justifies that modified finite maps are suitable for replacing references.

Constant cost of basic operations means that the number of keys during runtime has no effect on the performance of the basic operations unlike known finite maps implementation where the cost of basic operations is proportional to the logarithm of the number of used keys.

Thesis 2
Common Heap Interface is Definable

I defined a general heap interface which handles remote creation of references, heterogeneous and homogeneous heaps, shared references and heap unions and I compared the capabilities of six different heap models for the interface.

Paper [10]

The heap is the part of memory which can described as a set of memory location–value pairs. The heap can be seen as a non-persistent finite map too. References can be modelled by heaps in purely functional languages if the type system of the languages supports uniqueness typing.

- I defined a general heap interface. The interface handles remote creation of references, heterogeneous and homogeneous heaps, shared references and heap unions.
- I defined 6 different heap models for the general interface and I compared them.
- I showed that interface of heaps used in the Clean language can be replaced by the general interface, and the replaced interface correct avoids a severe bug in the original interface.

Thesis 3
Equality Check by References can be done Safely

I described a safe way to compare algebraic data by references.

Paper [5], related papers [3] and [4]

Compiled algebraic data type values contains references: every constructor has references pointing to its parameters. During pattern matching these references are read, but the comparison and changing of these references is unsafe because it ruins referential transparency.

I extended a purely functional language with reference comparison such that there is an easy precondition for referential transparency and unfoldability of functions holds unconditionally, unlike the solution in ν-kalkulus[15]. I have proved these assertions.

Related Publications

Journal papers [9], [10].

Papers in referred proceedings [5], [6].

Abstracts [3], [4], [7], [8].

References

[1] Stephen Adams. Efficient sets: a balancing act. *Journal of Functional Programming 3(4)*, pages 553–562, October 1993.

[2] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[3] Péter Diviánszky. Direct graph manipulation in functional languages. Presented at the Central-European Functional Programming School, Eötvös Loránd University, Budapest, Hungary, 4-16 July, 2005.

[4] Péter Diviánszky. Substructural functional programming. In A. Butterfield, editor, *Implementation and Application of Functional Languages, 17th International Workshop, IFL'05*, page 3 pages. Trinity College, Dublin, 2005. Technical Report TCD-CS-2005-60.

[5] Péter Diviánszky. Unique identifiers in pure functional languages. In Henrik Nilsson, editor, *Proceedings of the Seventh Symposium on Trends in Functional Programming (TFP)*, pages 84–98. The University of Nottingham, 2006.

[6] Péter Diviánszky. Efficient implementation of linearly used finite maps. In Zoltán Horváth, László Kozma, and Viktória Zsók, editors, *Proceedings of the 10th Symposium on Programming Languages and Software Tools*, pages 199–213. Eötvös University Press, 2007.

[7] Péter Diviánszky. Linearly used finite maps. In *2nd Central European Functional Programming School (CEFP 2007), Cluj Napoca, Romania, PhD workshop abstracts*, page 1 page, 2007.

[8] Péter Diviánszky. Translating imperative algorithms to functional code with unique variable environments. In Sven-Bodo Scholz, editor, *Implementation and Application of Functional Languages, 20th International Symposium, IFL*

2008, pages 216–221, Hatfield, Hertfordshire, UK, September 2008. Technical Report No. 474, School of Computer Science, University of Hertfordshire.

[9] Péter Diviánszky. Efficient implementation of linearly used finite maps. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30:201–218, 2009.

[10] Péter Diviánszky. Non-monadic models of mutable references. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Third Summer School, CEFP 2009, Budapest, Hungary, May 2009 and Komárno, Slovakia, May 2009, Revised Selected Lectures*, volume 6299 of *Lecture Notes of Computer Science*, pages 147–187. Springer Verlag, 2010.

[11] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell.

[12] B. Lippmeier. *Type inference and optimisation for an impure world*. PhD thesis, Australian National University, 2009.

[13] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[14] Harald Sondergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Inf.*, 27(6):505–517, January 1990.

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

[16] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast: A functional semantics for the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 25–36, New York, NY, USA, 2007. ACM.

[17] Andrew K. Wright. Typing references by effect inference. In Bernd Krieg-Brückner, editor, *ESOP 92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer, 1992.