MPRI course 2-4
"Programmation fonctionnelle et systèmes de types"
Programming project

Pierre-Évariste Dagand

2019–2020

Setting the scene

While visiting the University of Glasgow (Scotland), you find yourself stranded in the McMillan Reading Room. It is 4pm, the sun is setting, rain is pouring down and there is no one in sight to rescue you. To take your mind off the ghost of Lord Kelvin, which keeps making unwarranted ectoplasmic appearances, you start browsing the University collections. Right next to Papyrus 22 stands an intriguing pile of dusty papers describing the "Mighty Simplifier" of the ~~Glasgow~~ Glorious Haskell Compiler. You set out to drive out the pesky spirit of Lord Kelvin by sprinkling a healthy dose of OCaml code over a few sheets of paper: you thus implement an explicitly-typed System F, its type-checker and a transformation-based optimizer.

1 Summary

The purpose of this programming project is to implement a type-preserving optimizer from μ**Core** – a subset of the **Core** language of GHC [Peyton Jones et al., 1996] – onto itself. μ**Core** is a pure (*i.e.* free of side-effect) and total [Turner, 2004] (*i.e.* strongly normalizing) polymorphic, second-order lambda calculus, which amounts to Girard's System F extended with algebraic datatypes:

$$\textsc{Var}\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{Con}\ \frac{\mathrm{typeof}(K) = \forall \vec{X}.\vec{\sigma} \to T\ \vec{X} \qquad \overrightarrow{\Gamma \vdash u : \sigma[X \mapsto \varphi]}}{\Gamma \vdash K\ \vec{\varphi}\ \vec{u} : T\ \vec{\varphi}}$$

$$\textsc{Abs}\ \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \mathtt{fun}\ (x : T_1) = t : T_1 \to T_2} \qquad \textsc{TAbs}\ \frac{\Gamma, X \vdash t : T}{\Gamma \vdash \mathtt{fun}\ [X] = t : \forall X.T} \qquad \textsc{App}\ \frac{\Gamma \vdash t : T_1 \to T_2 \qquad \Gamma \vdash u : T_1}{\Gamma \vdash t\ u : T_2}$$

$$\textsc{TApp}\ \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t\,[\,\varphi\,] : T[X \mapsto \varphi]} \qquad \textsc{VBind}\ \frac{\Gamma \vdash u : T_1 \qquad \Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \mathtt{let}\ x = u\ \mathtt{in}\ t : T_2}$$

$$\textsc{Case}\ \frac{\overrightarrow{\mathrm{typeof}(K) = \forall \vec{X}.\vec{\sigma} \to T\ \vec{X}} \qquad \Gamma \vdash t : T\ \vec{\varphi} \qquad \vec{\nu} = \overrightarrow{\vec{\sigma}[X \mapsto \varphi]} \qquad \overrightarrow{\Gamma, \overrightarrow{x : \nu} \vdash u : \tau} \qquad \mathrm{ctors}(T) = \{\vec{K}\}}{\Gamma \vdash \mathtt{match}\ t\ \mathtt{return}\ \tau\ \mathtt{with}\ \overrightarrow{K\ \vec{x} \to u} : \tau}$$

We consider a simplistic execution model for this language, assuming call-by-need evaluation:

- `let` bindings – and *only* `let` bindings – require costly heap allocation, producing a thunk ;[1]
- pattern-matching – and *only* pattern-matching – triggers evaluation.

The optimizer thus strives to minimize the number of `let` bindings as well as the number of `match`, while also fulfilling the usual duty of an optimizer: performing partial evaluation (within reason) as well as dead-code elimination. Finally, a tacit expectation is for the optimizer to avoid code size blow-up. It works as a source-to-source transformation: it expects a well-typed μ**Core** program from which it ought to produce another well-typed μ**Core** program. On paper [Peyton Jones et al., 1996, Santos, 1995, Peyton Jones and Marlow, 2002, Maurer et al., 2017], it is simply specified through a collection of equations. The role of the Mighty Simplifier is to turn such crisp mathematical specifications into a potent optimizer, selectively rewriting the source program through a careful choice and orientation of the allowed equations.

Because of the open-ended nature of this exercise, the quality of your work will also be judged based upon your ability to provide examples illustrating – and explaining ! – good *as well as* bad behaviors of your optimizer. Conversely, our test cases exercising the optimization passes (Task 2 and Task 3b) are not prescriptive: you are free to improve upon our results. And, indeed, some of our test cases document some opportunities missed by our reference implementation.

This programming project builds upon Santos' PhD thesis [Santos, 1995] as well as some of its follow-up papers [Peyton Jones et al., 1996, Peyton Jones and Santos, 1998, Peyton Jones and Marlow, 2002], dating back to the early days of the Haskell project. A technical device introduced at the time – join points – has recently gathered renewed interest [Maurer et al., 2017, Cong et al., 2019] as it relates to the long-standing debate on using continuations as an intermediate language.

2 Required software

To use the sources that we provide, you need OCaml and Menhir. Any reasonably recent version should do. You also need the OCaml packages `pprint` and `ppx_deriving`. If you have installed OCaml via `opam`, issue the following command:

```
opam install menhir pprint ppx_deriving
```

3 Overview of the provided sources

Many components of the compiler are provided, including: definitions of the syntax of terms; a lexer and parser; a pretty-printer; some code for dealing with names and binders.

In the `src/` directory, you will find the following files:

pllib/identifierChop.mll, pllib/identifier.{ml, mli} An identifier is essentially a pair of a sort and a string. Each sort defines a disjoint namespace.

pllib/atom.{ml, mli} An atom is the internal object used to represent a name. It is a pair of a unique integer identity and a (not necessarily unique) string. The function `Atom.fresh` creates a fresh atom.

pllib/error.{ml, mli} Generic error reporting facilities.

pllib/import.{ml, mli} Generic facilities for converting identifiers to atoms and detecting unbound identifiers.

[1] Data constructors and lambda abstractions, being in weak head normal form, come essentially for free.

pllib/export.{ml, mli} Generic facilities for converting atoms back to identifiers, while avoiding unintentional capture.

pllib/lexerUtil.{ml, mli} Generic utilities for lexical analysis.

syntax.ml Abstract syntax for the types and terms of μ**Core**. This syntax is produced by the parser. In this version of the syntax, names are represented as identifiers. Here, there are three sorts of identifiers, for term variables, type variables, and record labels.

parser.mly, lexer.mll Together, the lexer and parser define the concrete syntax of the surface language.

types.{ml, mli} Internal representation for the types of μ**Core**, up to α-conversion. This representation is used directly by the type-checker and translator.

terms.{ml, mli} Internal representation for the terms of μ**Core**. This representation is used directly by the type-checker. It also serves as both the source and target languages of the simplifier. In this representation, all atoms are unique – that is, a term variable or a type variable is never bound twice in different places. This assumption can be exploited in the type-checker and simplifier. Hence, this invariant must be carefully preserved when generating new terms. Parameters of functions are explicitly annotated, so that type-checking is decidable.

internalize.{ml, mli} Checks that all identifiers are bound, and replaces identifiers with unique atoms, so as to produce an internal representation of types and terms. This code is executed after the parser.

print.{ml, mli} Pretty-printers for types and terms.

typerr.ml Auxiliary functions that help report type errors during type-checking.

tsubst.{ml, mli} A data structure for representing type substitutions (map from type variables to types). These are used in the type-checker and the simplifier to accumulate instantiated type variables and discharge them on-demand.

typecheck.{ml, mli} A type-checker for μ**Core**. This file is incomplete.

simplifier.{ml, mli} A transformation-based optimizer for μ**Core**. The provided skeleton implements a (convoluted) identity function.

main.ml This driver interprets the command line and invokes the above modules as required.

Makefile, _tags Build instructions. Issue the command "`make`" in order to generate the executable.

joujou The executable file for the program. Type "`./joujou` *filename*" to process the program stored in *filename*. This type-checks the expression, optimizes it while type-checking the result of each simplification pass to ensure that types are indeed preserved all along.

Testing In the `test/` directory are small programs written in μ**Core**, which you can give as arguments to `joujou`. In order to test your implementation, run "`make test`". The script submits the files `test/*.f` and `test/*.fj` to your compiler. When testing your simplifier, it (textually) compares your result with a suggested, non-prescriptive solution specified in `test/*.spec2` for Task 2 and `test/*.spec3` for Task 3b.

The files `test/*.fjrec` are not part of the test suite. They are written in an extension of μ**Core** supporting both join points (Task 3a) and recursive definitions (`let rec` and `join rec`).

4 Task description

Task 1 In the file **typecheck.ml**, implement the type-checker for the explicitly-typed language μ**Core**.

Test At this point, "`make test1`" should work.
Feel free to add more test files in the subdirectory `test/`.

Task 2 In the file **simplifier.ml**, implement the transformations (β), (β_τ), *(inline)*, *(drop)* and *(case)* specified in Maurer et al. [2017, Figure 4]. These transformations must be type-preserving. In the mandatory part of this project, we adopt a simplistic inlining strategy: we shall apply the rule *(inline) systematically*, ignoring the potential impact of inlining on code size.

Test At this point, "`make test2`" should work.
Feel free to add more test files in the subdirectory `test/`.

Task 3a Extend the μ**Core** language by adding two new primitive constructs, "`join` $j\ [\vec{X}]\,\vec{x} : \sigma = t$ `in` u" and "`jump` $j\ [\vec{\varphi}]\,\vec{t} : \sigma$", which amounts to defining a non-returning function, for the former, and calling such a function, for the latter [Maurer et al., 2017, §3]. We call the resulting language μ**CoreJ**. This language is specified in Figure 1 and Figure 2 of Maurer et al. [2017], which is a super-set of the language μ**Core** specified above. This requires **extending all compiler passes**, beginning with the lexer and parser, all the way down to the type-checker and simplifier.

Task 3b In the files **simplifier.{ml, mli}**, implement a case-of-case transformation. The motivation is to turn expressions of the form

$$\texttt{match}\ \left(\texttt{match}\ t\ \texttt{return}\ T\ \vec{\varphi}\ \texttt{with}\ \overrightarrow{K\ \vec{x} \to u}\right)\ \texttt{return}\ \sigma\ \texttt{with}\ \overrightarrow{alt}$$

into the more efficient form

$$\texttt{match}\ t\ \texttt{return}\ \sigma\ \texttt{with}\ \overrightarrow{K\ \vec{x} \to \texttt{match}\ u\ \texttt{return}\ \sigma\ \texttt{with}\ \overrightarrow{alt}}$$

while using a join point to share the context `match` [] `return` σ `with` $\overrightarrow{alt}$ across each of the branches so as to avoid code size blow-up. One can draw some inspiration from the transformation *(commute)* described in Maurer et al. [2017, §3]. Once again, this transformation must be type-preserving.

Test At this point, "`make test3`" should work.
Feel free to add more test files in the subdirectory `test/`.

Optional tasks Having successfully implemented the above tasks will give you in a very good grade. Nonetheless, if you wish to go further and receive extra credit, there are a number of things that you might do. Here are some *suggestions*. This list is not sorted and not limiting. Not all suggestions are easy: we expect a good student to be able to implement one such task whereas a very strong student might be able to put together two or three optimization tasks while showcasing interesting interactions between these optimizations. A contractually non-binding indication of the difficulty of each tasks is indicated by the number of $\star$: the more, the merrier. Think before attacking an ambitious extension!

- $\star$ Add recursive `let` and `join` (denoted by `let rec` and `join rec` in the grammar) to the language. Extend the simplifier to support such definitions. Note that, by introducing non-termination, these constructions modify the notion of program equivalence, which ought to be respected by the simplifier. A few sample programs, recognizable by their extension `.fjrec`, are provided.

⋆ Optimizers allow programmers to write idiomatic code – aimed at fellow humans – safe in the knowledge that the optimizer will produce code that runs efficiently on a computer. Sadly, optimizers are complex beasts: to improve legibility of the compilation process, industrial-strength compilers provide various mechanisms to improve traceability, be it "remark diagnostics" in LLVM [LLVM dev. team], rule "dumps" and "checks" in GHC [GHC dev. team] or "inlining reports" in OCaml [OCaml dev. team]. Augment your optimizer to provide *actionable* feedback to the programmer. Be sure to demonstrate your tool on actual programs, showing in particular its behavior on programs that appear to be insufficiently optimized.

⋆⋆⋆ Our inlining strategy outlined above is extremely simplistic: in practice, it would cause significant code duplication, so much so as to be unusable. Propose and implement a more fine-grained inlining strategy that would avoid such code size blow-up. One can for example draw inspiration from Peyton Jones and Marlow [2002].

⋆⋆⋆ The potency of the simplifier is sometimes limited by the presence of `let` or `join` bindings that may stand in the way of a crucial simplification. In GHC, two additional optimization passes are present in the compiler pipeline [Santos, 1995]: a float-in pass – where bindings are pushed as deeply toward the leafs of case trees as possible – and a float-out pass – where bindings are pushed as far up the top-level as possible. Add either or both of these and evaluate their effect.

⋆⋆ Our simplifier currently performs simplifications justified by the equational theory of μ**Core**. However, when implementing high-level libraries, more abstract equational theories arise. For instance, the OCaml programmer knows that `List.map (fun x -> x)` is equal to the identity, or that `List.map f (List.map g xs)` can be simplified to `List.map (fun x -> f (g x)) xs`, which avoids an extra iteration. However, the compiler, unaware of `List.map` is unable to perform these simplifications on its own. Extend μ**Core** and its simplifier with "rewrite rules" [Peyton Jones et al., 2001] and demonstrate their benefits.

⋆ Join points are a conservative extension of μ**Core** [Maurer et al., 2017, §6]: implement a type-preserving erasure of μ**CoreJ** to μ**Core**.

⋆⋆⋆ A key benefit of join points is that they provide a construct to express non-returning control-flow, with special support from the compiler back-end (avoiding unnecessary heap allocation). In such a language, certain `let`-definitions can be automatically promoted into `join`-definitions, a processed known as "contification" in the literature [Kennedy, 2007]. Join points enable an inexpensive and lightweight form of contification, as specified in Maurer et al. [2017, §4]. Add contification to your simplifier and demonstrate its benefits.

In each case, please write **a textual explanation** of what you did, how you did it, and where to look for it in your code. Also, propose **test files** that illustrate what you did.

5 Evaluation

Assignments will be evaluated by a combination of:

- **Testing**. Your compiler will be tested with the input programs that we provide (make sure that "`make test`" succeeds!) and with additional input programs.
- **Reading**. We will browse through your source code and evaluate its correctness and elegance.

6 What to turn in

When you are done, please e-mail to Pierre-Évariste Dagand, François Pottier, Yann Régis-Gianas and Didier Rémy a `.tar.gz` archive containing:

- All your source files.
- Additional test files written in the small programming language, if you wrote any.
- If you implemented "extra credit" features, a `README` file (written in French or English) describing these additional features, how you implemented them, and where we should look in the source code to see how they are implemented.

7 Deadline

Please turn in your assignment on or before **February 21, 2020**.

References

Y. Cong, L. Osvald, G. M. Essertel, and T. Rompf. Compiling with continuations, or without? whatever. *PACMPL*, 3(ICFP):79:1–79:28, 2019. doi:10.1145/3341643.

GHC dev. team. Core representation and simplification. https://downloads.haskell.org/ghc/latest/docs/html/users_guide/debugging.html#core-representation-and-simplification.

A. Kennedy. Compiling with continuations, continued. In *ICFP*, pages 177–190, 2007. doi:10.1145/1291151.1291179.

LLVM dev. team. Remarks. https://llvm.org/docs/Remarks.html.

L. Maurer, P. Downen, Z. M. Ariola, and S. L. Peyton Jones. Compiling without continuations. In *PLDI'17*, pages 482–494, 2017. doi:10.1145/3062341.3062380.

OCaml dev. team. Inlining reports. https://caml.inria.fr/pub/docs/manual-ocaml/flambda.html#sec500.

S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, September 2001.

S. L. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell compiler inliner. *J. Funct. Program.*, 12(4&5):393–433, 2002. doi:10.1017/S0956796802004331.

S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for haskell. *SCP*, 32(1-3): 3–47, 1998. doi:10.1016/S0167-6423(97)00029-4.

S. L. Peyton Jones, W. Partain, and A. L. M. Santos. Let-floating: Moving bindings to give faster programs. In *ICFP'96*, pages 1–12, 1996. doi:10.1145/232627.232630.

A. Santos. *Compilation by transformation for non-strict functional languages*. PhD thesis, July 1995. URL http://theses.gla.ac.uk/id/eprint/74568.

D. A. Turner. Total functional programming. *JUCS*, 10(7):751–768, 2004. doi:10.3217/jucs-010-07-0751.