Experiences in developing a compiler for *Safe* using Haskell

Manuel Montenegro [1,2] Ricardo Peña [1,3] Clara Segura[1,4]

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Abstract

Safe is an eager language introduced as a research platform for programming small devices and embedded systems with strict memory requirements. It follows a semi-explicit approach to memory control combining regions and a deallocation construct but with a very low effort from the programmer's point of view. Here we describe our experiences in implementing a compiler for *Safe* using Haskell. We show how polymorphism, higher-order functions, monads and different kinds of libraries have been useful in the implementation of all compiler phases.

Keywords: Memory management, compiler implementation.

1 Introduction

Safe[5] was introduced as a research platform for investigating the suitability of functional languages for programming small devices and embedded systems with strict memory requirements. The final aim is to infer, at compile time, safe upper bounds on memory consumption. The compiler produces Java bytecode so *Safe* programs can be executed in most mobile devices and web navigators.

In most functional languages memory management is delegated to the runtime system. Fresh heap memory is allocated during program evaluation as long as there is enough free memory available. Garbage collection interrupts program execution in order to copy or mark the live part of the heap so that the rest is considered as free. This does not avoid memory exhaustion if not enough free memory is recovered to continue execution. The main advantage of this approach is that programmers do not have to bother about low level details concerning memory management.

[1] Work supported by the projects TIN2008-06622-C03-01/TIN (STAMP), S-0505/TIC/0407 (PROMESAS)
[2] Email: montenegro@fdi.ucm.es. Work supported by the MEC FPU grant AP2006-02154.
[3] Email: ricardo@sip.ucm.es
[4] Email: csegura@sip.ucm.es
[5] Available at: http://dalila.sip.ucm.es/safe

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Its main disadvantages are the time delay introduced by garbage collection, the personal or economic damages provoked by memory exhaustion, and the difficulty of reasoning about memory consumption. These reasons make automatic memory management unacceptable in small devices where garbage collectors are a burden both in space and in service availability. Programmers of such devices would like both to have more control over memory and to be able to reason about the memory consumption of their programs.

Our first-order functional language *Safe* is a semi-explicit approach to memory control which combines regions and a deallocation construct. Implicit regions are used to destroy garbage. They are allocated/deallocated by following a stack discipline associated to function calls and returns. Each function call allocates a local working region, which is deallocated when the function returns. The compiler infers which data structures may be allocated in this local region because they are not needed as part of the result of the function [9]. In order to overcome the problems related to nested regions, *Safe* provides a **case!** construct that deallocates the individual cells of a data structure, so that they can be reused by the memory management system. Regions and explicit destruction are orthogonal mechanisms: we could have destruction without regions and viceversa. This combination of explicit destruction and implicit regions is novel in the functional programming field. We have defined a type system [10] and a type inference algorithm [12] guaranteeing that none of the two mechanisms create dangling pointers in the heap.

Safe's syntax is a first-order subset of Haskell extended with destructive pattern matching, so programming in *Safe* is straightforward for Haskell programmers. They only have to write a destructive pattern matching, denoted by ! or a **case!** expression, when they want a cell to be reused. As an example, we show an append function destroying the first list's spine, while keeping its elements in order to build the result. Using recursion the recursive spine of the first list is deallocated:

```
appendD :: [a]! -> [a] -> [a]
appendD []!     ys  = ys
appendD (x:xs)! ys  = x : appendD xs ys
```

This version needs constant additional heap space (a cell is destroyed and another one is created at each call), while the usual one needs additional linear heap space. Type $[a]!$ denotes the type of a list being destroyed by the function.

In this paper we describe our experiences in implementing a compiler for *Safe* using Haskell. In Section 2 we describe the different phases of the compiler. In Section 3 we show how polymorphism, higher-order functions and monads provided by Haskell have been used in several phases. In Section 4 we mention the libraries we found useful for our implementation. In Section 5 we provide our conclusions.

2 A brief overview of the Safe's compiler

The phases of the compiler's frontend are shown in Fig. 1. First, the input is scanned and parsed in order to obtain an abstract syntax tree (AST), which is represented as a Haskell term of type `Prog a`, where `a` is a polymorphic decoration (Sec 3.1). We have used standard tools (Alex [2] and Happy [8]) to implement these phases (Sec. 4.1). Then, the following phases are performed:

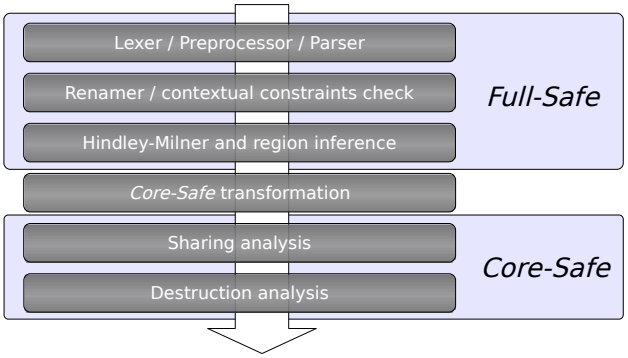

Fig. 1. *Safe*'s compiler frontend

- **Renamer / Contextual constraints check:** The analyses implemented in the subsequent phases assume that all bound variables have different names, so each variable is replaced by a fresh one. Additionally, we check the language's contextual constraints, e.g. that every variable is in scope, that functions and data constructors are called with the right number of arguments, etc.
- **Hindley-Milner type and region inference:** This phase decorates every expression, data constructor and function definition with its corresponding type, after inferring it. The AST is also decorated with information about the regions in which each data structure lives. In particular, it determines whether a given data structure is local or not to the current function call.
- *Core-Safe* **transformation:** The original *Safe* program is translated into a desugared and semantically equivalent *Core-Safe* version. The previously inferred type decorations are preserved.
- **Sharing analysis:** Given two variables belonging to the same function definition, it computes whether the respective data structures pointed to by them at runtime may share memory locations. The program is decorated with this sharing information [13], which is needed by the following phase.
- **Destruction analysis:** It infers a typing for the source program w.r.t. the type system in [10], guaranteeing that dangling pointers are not generated as a consequence of a **case**!.

The user can choose between different available backends. We have implemented the translation from *Core-Safe* to the language of the Safe Virtual Machine [11], which may be further translated into Java bytecode in order to produce platform-independent executables. A certificate written in Isabelle/HOL is also generated in order to prove the absence of dangling pointers at runtime. We have also implemented the generation of Term Rewriting Systems [7], which may be used in conjunction with existing tools for proving termination, such as AProVE or μ-Term.

3 Polymorphism, higher-order and monads

3.1 Polymorphic datatypes

The use of polymorphic datatypes is intensive along all the phases of the compiler. First, they are used to represent the abstract syntax of programs and expressions:

```
type Prog a = ([DataDec], [Def a], Exp a)  -- programs
...
data Exp a = ConstE Lit a                          -- literal
           | ConstrE String [Exp a] RegVar a        -- constructor application
           | VarE String a                          -- variable
           | CopyE String RegVar a                  -- copy expression
           | ReuseE String a                        -- reuse expression
           | AppE String [Exp a] [RegVar] a         -- function application
           | LetE [Def a] (Exp a) a                 -- let expression
           | CaseE (Exp a) [CaseAlt a] a            -- non-destructive case
           | CaseDE (Exp a) [CaseAlt a] a           -- destructive case
```

The polymorphic argument `a` is used to decorate the abstract syntax tree with different kinds of information:

- The Hindley-Milner inference phase decorates programs and expressions with their types:

  ```
  decorProg :: Assumps -> Prog a -> (Assumps, Prog TypeExp)
  ```

- The sharing analysis augments the decoration with information about sharing:

  ```
  sharingProg :: Prog TypeExp -> Prog (TypeExp, SharingDec)
  ```

- The destruction analysis phase decorates programs with information about the destructive nature of variables. When destructive pattern matching is not safely used by the programmer an error is returned and the program is rejected:

  ```
  destInferenceProg :: DestEnv -> Prog (TypeExp, SharingDec)
                               -> Either DestInferenceError (DestEnv, Prog DestDec)
  ```

Also, polymorphic types defined in libraries are instantiated in order to define many different types used in the compiler, in particular `Data.List`, `Data.Map`, and `Data.Set` where the standard functions for lists, maps from ordered keys to values (dictionaries), and ordered sets are available. Both sets and maps are efficiently implemented using size balanced binary trees. We use maps for defining:

- Environments containing information about functions and/or variables, such as sharing signatures, destruction signatures, types etc:

  ```
  type ShEnvironment = Map String ShSignature   -- sharing signatures for functions
  type Relations = Map Variable ShInfo          -- sharing information for variables
  type DestEnv = Map String DestSig             -- destruction signatures for functions
  type Assumps = Map String TypeExp             -- HM types
  ```

- Substitutions, such as region substitutions obtained during region inference and type substitutions obtained during type inference:

  ```
  type RegSubst = Map TypeVar TypeVar    -- region substitution
  type Subst = Map TypeVar TypeExp       -- type substitution
  ```

- Tables containing different kinds of auxiliary information, such as the number of region parameters in data declarations, the recursive positions of each constructor and the types of the recursive calls (Sec. 3.3):

  ```
  type DecDataInfo = Map String Int      -- number of region parameters of a data
  type TablaRecPos = Map String [Int]    -- recursive positions of a constructor
  type RecCalls = Map RecId TypeExp      -- types of recursive calls
  ```

We use in several places sets containing program or type variables. For example, the decoration generated by the destruction analysis consists basically of four sets of program variables (safe, condemned, in-danger and unknown [12]) :

```
type Variables = Set Variable
data DestDec = DD (Variables, Variables, Variables, Variables) ExpDepDec
```

We also use nested datatypes. For example, the sharing information is a map from variable names to a tuple of seven sets of variables (see [13] for details):

```
type Relations = Map Variable ShInfo
type ShInfo = (Variables, Variables, Variables,Variables, Variables, Variables, Variables)
```

3.2 Standard higher-order functions

Many standard higher-order functions available for lists, like folds and map, have been intensively used in the compiler, as well as their counterparts in other polymorphic data types, e.g. sets and maps. We highlight an accumulating map function, used almost in all the phases of the compiler, including code generation:

```
mapAccumL :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])
mapAccumL _ s []     =  (s, [])
mapAccumL f s (x:xs) =  (s'',y:ys)
                     where (s', y ) = f s x
                           (s'',ys) = mapAccumL f s' xs
```

It behaves as a combination of `map` and `foldl`: while applying the argument function to each element of the list, it passes an accumulator from left to right, which is returned as result together with the new list. This function has been useful whenever an environment or a state must be accumulatively threaded while processing a program, where different kinds of lists appear: function definitions, call arguments, case alternatives etc. We have found it useful to define a monadic version:

```
mapAccumM :: Monad m => (acc -> x -> m (acc,y)) -> acc -> [x] -> m (acc, [y])
mapAccumM f acc [] = return (acc, [])
mapAccumM f acc (x:xs) = do (acc',y) <- f acc x
                            (acc'',ys) <- mapAccumM f acc' xs
                            return (acc'', y:ys)
```

3.3 Error and State Monads. Use of Laziness

An extensive use of the Monad Transformer Library(`mtl`) is done. This library, inspired by [5], provides definitions of several monads and monad combinators. The State monad encapsulates computations requiring the explicit propagation of a state. It is used in the Hindley-Milner type and region inference phase. This phase traverses the AST of a given definition and decorates it with a preliminary type, while propagating an internal state, wrapped in the monad as follows:

```
type IntState = ([Equation],[Constraint], Set TypeVar, [TypeVar], [TypeVar],
                 [String], RecCalls)
type HMState = State IntState
```

During the AST traversal, several unification equations and constraints between types are generated and stored in the first two components of the state. This may involve the generation of fresh names for type and region type variables. These are represented as lazy infinite lists in the fourth and fifth components of the state. When one of these is needed, the head of the corresponding list is taken and its tail is put back into the state. The set of explicit generated region variables, needed by the region inference, is stored in the third component. In addition, the type assigned to each recursive call is uniquely identified and stored in a separate table of type `RecCalls` (seventh component), used later to infer polymorphic recursion over regions. The recursive call identifiers are generated in the same way as type variables.

As an example, the following (simplified) code fragment decorates a **let** expression. The state is propagated from the auxiliary expression to the main one. We compare below the monadic approach with an explicit state propagation approach:

Explicit propagation:

```
decorAndGenExp :: Assumps -> Exp a -> IntState -> (IntState, Exp a)
decorAndGenExp as (LetE defs exp _) st = (st2, LetE defs' exp' (decExp exp'))
    where (st1, (as',defs')) = decorAndGenInnerDefs as stdefs st
          (st2, exp') = decorAndGenExp (M.union as as') exp st1
```

State monad:

```
decorAndGenExp :: Assumps -> Exp a -> HMState (Exp TypeExp)
decorAndGenExp as (LetE defs exp _) = do (as',defs') <- decorAndGenInnerDefs as defs
                                         exp'        <- decorAndGenExp (M.union as as') exp
                                         return (LetE defs' exp' (decExp exp'))
```

A state-monadic computation is started by the function `runST`. It receives the monadic computation to be executed and the initial state, and returns the result of the computation (the decorated tree, in our case) together with the final state.

When dealing in Haskell with functions that may produce an error, there are several approaches. The most popular one is the `error` function, which allows little possibility of error handling. This can be solved by using the `Either a b` datatype, so that the caller can establish whether the computation failed, and what kind of error has taken place. Errors are handled in this fashion in the renaming phase, i.e. the corresponding function may return either an error or the modified program:

```
sparser :: Show a => Prog a -> Either SemanticError (Prog a)
```

where the `SemanticError` datatype provides information about the error thrown. The `mtl` library defines the datatype `Either a` as a monad, allowing to combine partial computations. In this phase, the `Either` monad is combined with the `State` monad, which propagates the arities of each type, constructor and function in the program:

```
data SemState = SemState { typeNames :: Map String Int, constrNames :: Map String Int,
                           funcNames :: Map String Int,...}
type Sem e a = StateT SemState (Either e) a
```

For example, the following code fragment checks a function application:

```
checkExp :: Exp a -> Sem DefError (Exp a)
checkExp (AppE f es rs dec) =
    do -- We look up the arity of the function in the state
       mar <- lookupT funcNames f
       -- Is the function f defined?
       ar <- maybe (throwError (ExpUndefinedId f)) return mar
       -- Is the number of arguments correct? Partial applications are not supported.
       when (ar /= length es) (throwError (HigherOrderApp f (length es) ar))
       -- Perform the renaming on the arguments and return the result.
       es' <- mapM checkExp es
       return (AppE f es' rs dec)
```

where the `maybe` function is given a value of type `Maybe` as its third parameter. If its value is `Just x`, the function passed as its second parameter is applied to `x` and the result returned. If its value is `Nothing`, the first parameter is returned. The `when` φ combinator executes a given action provided the condition φ is satisfied.

4 External libraries

4.1 Parsing and pretty printing tools

Haskell provides many different tools for helping in the initial phases of the compiler: lexical analysis or scanning, and syntactic analysis or parsing. We have chosen to use the scanner generator Alex [2] and the parser generator Happy [8].

The first one receives as input a regular grammar describing the lexical units —or *tokens*— of the input language, and produces as output a scanner written in Haskell which, when executed, scans the input language and, provided there are no lexical errors, produces a list of tokens as output. Alex gives facilities to annotate tokens with its position (line and column numbers) in the input text. This is useful for generating meaningful errors, but we also used the column positions to

implement the *layout rule* of some functional languages, included Safe: the starting column of the text has syntactic meaning, as it can be used to open a new declaration (by starting it in the same column as the previous one), to close the current declaration block (by moving some columns to the left), or to continue with the current declaration (by moving to the right).

The parser generator Happy receives as input an LALR-(1) grammar describing the input language, and produces as output a parser written in Haskell which, when executed, parses the list of tokens produced by the scanning phase and, provided there are no syntactic errors, produces the abstract syntax tree of the program being compiled. Our grammar has 31 terminal symbols, 43 non-terminal and 120 rules. The LALR-(1) automaton generated by Happy has 243 states.

We also use a pretty printing library in order to present the different intermediate files in an easy-to-read format. The one chosen is `PPrint` [6] by D. Leijen, based on the P. Wadler paper [14], based in turn on the famous J. Hughes combinator library [4]. This library is claimed to be 30% shorter and 30% faster than Hughes's one. It provides a class `Pretty` with the overloaded function `pretty :: Pretty a => a -> Doc` which creates pretty-printed versions of some simple types such as integers, booleans, tuples, and lists. All we had to do is overload the function `pretty` with the Haskell types describing our intermediate files. Most of these are differently decorated versions of the abstract syntax tree. Some other are the token list returned by the scanning phase, and the instruction lists returned by the different code generation phases. A number of combinators to indent, concatenate, group, etc., pretty documents is provided by the library to define the different instances of `pretty`.

4.2 Web interface support

We are building a web-based version of the compiler in which a remote user may interactively activate each compiler's phase and browse or change the intermediate files produced. This compiler generates XML versions of the files so that the user may browse them in a web browser. To this aim, we use the library `HaXml`, [6] and other related tools such as `DrIFT` [15] and `polyparse`, [7] which provide support for translating Haskell types into XML, and vice versa. This version of the compiler, each time it is invoked, only executes a given phase. All of them expect and produce XML files.

5 Conclusions

Summarising our experiences in using Haskell for building the *Safe*'s compiler, we believe that that Haskell has reached a high level of maturity, both from the point of view of its offered constructs, and from its available tools such as the GHC compiler, the various libraries, and other related programs. Without the use of higher-order functions (such as `mapAccumL, zipWith, map, foldr, ...`) and polymorphism, the *Safe* compiler would have been much more longer and painful to build. The rest of language features, such as laziness, monads, type classes, etc., have contributed in

[6] Available at: http://www.cs.york.ac.uk/fp/HaXml/
[7] Available at: http://www.cs.york.ac.uk/fp/polyparse/

a positive way to specific parts of the compiler. The specialised libraries for scanner and parser generation, and for web-interfacing have also saved a lot of work.

Perhaps, we have missed a better aid for debugging. The official supported tool in the ghc library is the function `trace`, which is very primitive. More sophisticated tools such as Hat [1] and Hood [3] are supported for earlier versions of GHC, but not for the current ones (we have used ghc 6.10). It is a pity that such useful tools are not adequately maintained in order to conform to the latest GHC versions.

Our development has spanned four years of non-continuous work, around ten people have participated in different parts of the compiler, and this is still growing. The current figures are: about 20.000 lines of code, including comments, distributed among 40 Haskell modules, some of them automatically generated by other tools.

A last remark is that the module called `AbstractSyntax` has been very useful in achieving good compiler modularity. The abstract syntax tree constitutes the main interface between most compiler phases, as many of them need a syntax tree either as input, or as output, or both. This has allowed us to work in parallel in different modules without major interferences between programmers.

References

[1] O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for tracing. In *14th International Workshop on Implementation of Functional Languages, IFL'02*, pages 165–181. LNCS 2670, 2003.

[2] C. Dornan, I. Jones, and S. Marlow. *Alex User Guide*, 2007. Available at: http://haskell.org/alex/.

[3] A. Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Haskell, Technical report of the University of Nottingham*, 2000.

[4] J. Hughes. The design of a pretty-printer library. In *1st International Spring School on Advanced Functional Programming Techniques, AFP'95*, pages 53–96. LNCS 925, 1995.

[5] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *1st International Spring School on Advanced Functional Programming Techniques, AFP'95*, pages 97–136. LNCS 925, 1995.

[6] D. Leijen. *PPrint, a prettier printer*, 2001. Available at: http://legacy.cs.uu.nl/daan/pprint.html.

[7] S. Lucas and R. Peña. Rewriting Techniques for Analysing Termination and Complexity Bounds of SAFE Programs. In *Draft Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'08*, pages 43–57, 2008.

[8] S. Marlow and A. Gill. *Happy User Guide*, 2001. Available at: http://haskell.org/happy/.

[9] M. Montenegro, R. Peña, and C. Segura. A Simple Region Inference Algorithm for a First-Order Functional Language. In *Draft Proceedings of the 9th Symposium on Trends in Functional Programming, TFP'08*, pages 194–208, 2008.

[10] M. Montenegro, R. Peña, and C. Segura. A type system for safe memory management and its proof of correctness. In *10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP'08*, pages 152–162, 2008.

[11] M. Montenegro, R. Peña, and C. Segura. A Resource-Aware Semantics and Abstract Machine for a Functional Language with Explicit Deallocation. In *Selected papers of the 17th International Workshop on Functional and (Constraint) Logic Programming, WFLP'08. To appear in ENTCS (15 pages)*, 2009.

[12] M. Montenegro, R. Peña, and C. Segura. An inference algorithm for guaranteeing safe destruction. In *Selected papers of the 18th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'08, LNCS 5438*, pages 135–151, 2009.

[13] R. Peña, C. Segura, and M. Montenegro. A Sharing Analysis for SAFE. In *Selected Papers of the 7th Symp. on Trends in Functional Programming, TFP'06.*, pages 109–128. Intellect, 2007.

[14] P. Wadler. A prettier printer. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 223–244. Palgrave Macmillan, 2003.

[15] N. Winstanley and J. Meacham. *DrIFT user guide. Version 2.2.3*, February 2008. Available at: http://repetae.net/computer/haskell/DrIFT/.