National and Kapodistrian University of Athens
Department of Mathematics
Graduate Program in Logic and Theory of Algorithms and Computation

An efficient implementation of lazy functional programming languages based on the generalized intensional transformation

M.SC. THESIS

PANAGIOTIS THEOFILOPOULOS

Supervisor : Nikolaos S. Papaspyrou
Associate Professor N.T.U.A.

Athens, December 2013

Η παρούσα Διπλωματική Εργασία

εκπονήθηκε στα πλαίσια των σπουδών

για την απόκτηση του **Μεταπτυχιακού Διπλώματος Ειδίκευσης**

στη

Λογική και Θεωρία Αλγορίθμων και Υπολογισμού

που απονέμει το

Τμήμα Μαθηματικών

του

Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών

Εγκρίθηκε την 23η Δεκεμβρίου 2013 από Εξεταστική Επιτροπή αποτελούμενη από τους:

	Ονοματεπώνυμο	**Βαθμίδα**	**Υπογραφή**
1.	Νικόλαος Παπασπύρου	Αν. Καθηγητής Ε.Μ.Π.	.....
2.	Παναγιώτης Ροντογιάννης	Αν. Καθηγητής Ε.Κ.Π.Α.	.....
3.	Ιωάννης Σμαραγδάκης	Αν. Καθηγητής Ε.Κ.Π.Α	.....

Panagiotis Theofilopoulos
Electrical and Computer Engineer

Περίληψη

Αυτή η εργασία διερευνά θεωρητικά και πρακτικά ζητήματα της αλληλεπίδρασης μεταξύ (ευρέως γνωστών και νέων) τεχνικών μεταγλώττισης, όπως ο γενικευμένος νοηματικός μετασχηματισμός, το defunctionalization, η ξεχωριστή μεταγλώττιση και το lambda lifting.

Ένας πειραματικός μεταγλωττιστής για τη γλώσσα Haskell (GIC), ο οποίος χρησιμοποιεί τις τεχνικές αυτές, δίνει τη δυνατότητα σε νέες ιδέες να υλοποιηθούν και να αξιολογηθούν μέσα σε ένα πρακτικό πλαίσιο.

Ως μέρος αυτής της δουλειάς πραγματοποιήθηκαν διάφορες προσθήκες και αλλαγές στο μεταγλωττιστή, είτε προκειμένου να γίνει ο μεταγλωττιστής πληρέστερος είτε προκειμένου να βελτιωθεί ο τελικός κώδικας που παράγεται από το LAR back-end του μεταγλωττιστή.

Λέξεις κλειδιά

Νοηματικός μετασχηματισμός, ξεχωριστή μεταγλώττιση, defunctionalization, οκνηρή εγγραφή ενεργοποίησης, lambda lifting, Haskell.

Abstract

This dissertation investigates theoretical and practical issues of the integration between (well-known and novel) compilation techniques, such as the generalized intensional transformation, defunctionalization, separate compilation, and lambda lifting.

An experimental Haskell compiler (GIC), which incorporates these techniques, serves as a workbench allowing ideas to be demonstrated and evaluated in a practical context.

Within the scope of this work, several additions and changes were made to the compiler either towards enchancing the tool's robustness or towards the optimization of the code emitted by the compiler's LAR back-end.

Key words

Intensional transformation, separate compilation, defunctionalization, lazy activation record, lambda lifting, Haskell.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή Νικόλαο Παπασπύρου για την καθοδήγηση που μου παρείχε για την πραγμάτωση αυτής της εργασίας, για την υποστήριξη του σε όλες της φάσεις της υλοποίησής της, τις πολύτιμες γνώσεις που μου μετέδωσε κατά τη διάρκεια τόσο των προπτυχιακών όσο και των μεταπτυχιακών μου σπουδών και τις πολλαπλές αφορμές που μου έδωσε να προσεγγίσω το αντικείμενο ενασχόλησής μου για το εγγύς μέλλον.

Ευχαριστώ τον συνεπιβλέποντα καθηγητή Παναγιώτη Ροντογιάννη για την προτροπή του να αναλάβω τη συγκεκριμένη εργασία και την ενθάρρυνσή του, το έντονο ενδιαφέρον που μου δημιούργησε για θέματα αυτού του είδους και τις πολύτιμες, νέες για εμένα, γνώσεις που μου μετέδωσε. Επίσης, ευχαριστώ το διδακτορικό φοιτητή Γεώργιο Φουρτούνη, με τον οποίο είχα μια εξαιρετική συνεργασία στο πλαίσιο της παρούσας δουλειάς, για την ουσιαστική και έμπρακτη βοήθεια που μου παρείχε συνεχώς καθώς και για τις χρήσιμες και ενδιαφέρουσες συζητήσεις μας πάνω στα σχετικά τεχνικά θέματα.

Τέλος, ευχαριστώ την οικογένεια και τους φίλους μου για τη συμαντική έμπρακτη υποστήριξή τους σε όλη τη διάρκεια των μαθητικών και φοιτητικών μου χρόνων.

Παναγιώτης Θεοφιλόπουλος,

Αθήνα, 23η Δεκεμβρίου 2013

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-2-13, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Δεκέμβριος 2013.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Acknowledgements

I would like to thank my advisor Nikolaos Papaspyrou for the guidance he provided for the realization of this dissertation, for his support regarding all the stages of the corresponding implementation, for the valuable knowledge that he imparted to me during both my undergraduade and postgraduate studies, and for the multiple initiatives he provided enabling me to approach the subject of my research for the near future.

I thank my co-advisor Panagiotis Rondogiannis for urging me to take up this work, for stimulating my strong interest for subjects of this kind and for the valuable, new to me, knowledge he imparted to me. Also, I thank the PhD student Georgios Fourtounis, with whom I had an excellent cooperation within the scope of this work, for the essential and factual help he had been constantly providing me, and for all the interesting and useful conversations we had on related technical subjects.

Finally, I thank my family and friends for their important, factual support throughout my school and university years.

Panagiotis Theofilopoulos,

Athens, December 23, 2013

This thesis is also available as Technical Report CSD-SW-TR-2-13, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, December 2013.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Figures

Chapter 1

Introduction

1.1 Purpose

In this dissertation we experiment with the integration of well-known and novel compilation techniques for lazy functional languages. Haskell serves as a concrete example of such a language. This work builds upon an existing experimental Haskell compiler, GIC, which incorporates two novel ideas: The generalized intensional transformation, which is used to transform programs in a first-order functional language to programs in a zero-order language with intensional operators, and defunctionalization in a separate-compilation setting. In its present form, GIC (its LAR back-end in particular) uses low-level C as the target language.

The project aims generally at developing a competitive Haskell compiler that can be considered as a serious alternative to existing compilers (either on its own or as a back-end) and also at developing, evaluating, and demonstrating new compilation techniques for non-strict functional languages.

Some initial directions of this particular dissertation are summarized below:

- Investigate the possibility of completely eliminating the C stack in order to obtain better runtime performance, easier garbage collection, and possibly easier implementation of optimizations at the C level (LAR back-end).
- Investigate possible solutions for the problem of garbage collection (LAR back-end).
- Tweak the code generator in order to produce faster-executing and/or more memory-efficient C code (LAR back-end).
- Whenever possible, add missing features.

1.2 Motivation

The execution of programs in lazy functional languages traditionally involves techniques derived from graph reduction [Wads71]. Our compiler takes a different approach, combining old and new techniques.

Defunctionalization [Reyn72] is used to eliminate all higher-order functions, effectively solving the problem of handling higher-order expressions and significantly reducing the complexity of all back-ends, which have to handle only a first-order lazy language. Although defunctionalization has been widely considered as a whole-program compilation technique, here it is adapted to allow for separate compilation.

The compiler's LAR back-end in particular relies on the generalized intensional transformation which supports non-strictness in a natural way. Lazy activation records (LARs) complement the generalized intensional transformation and fill in the details towards the compilation to low-level (currently C) code by representing function activation records and data values in a uniform way, providing an efficient representation for unevaluated expressions, and also providing support for efficient case analysis on data structures.

The new approach seems to be a promising one in several ways:

- The design is modular and flexible. The stages of compilation are independent, simple, well-understood and, in most cases, well-tested in many different contexts (i.e., other than our project) during long time periods. As a result, most additions and changes are relatively effortless, therefore allowing the quick development of the project by very few people.

- The compiler, especially when using the LAR back-end, is competitive in terms of execution speed of the compiled programs. Avoiding the deployment of any kind of abstract machine altogether along with the utilization of the LAR intermediate representation makes the back-end easily retargetable, while the generated machine code is very likely to remain efficient regardless of the particular (low-level) language used as the back-end's target. This allows for much flexibility in the code generator's design, as the designer can choose which specific features (for example calling convention, garbage collection, memory organization, etc.) are to be implemented from scratch and which are to take advantage of the facilities provided by the chosen target language.

- Although execution speed can already be considered competitive, there is probably still much room for further improvement. The compiler currently has a simple and incomplete front-end which performs almost no optimizations at all. Similarly, the LAR back-end's code generator relies mostly on the generalized intensional transformation performing only simple, mostly low-level, optimizations itself. Characteristically, simple optimizations that were implemented and introduced to the back-end as a part of the present dissertation lead to significant gains in terms of execution speed. All the above observations together with some existing ideas not implemented yet and the fact that the novel techniques used in the project seem to leave much unexplored space seem to indicate that further execution speedup can be expected in the near future. In other words, it seems that we have not yet made the most out of the new design and, taking into account the already satisfying performance, this is probably good news: there seems to be a potential for this compiler to soon become a high-performance one.

- Apart from the possible practical advantages mentioned above, there is also another benefit derived from this project: the compiler, along with a set of interpreters for the various intermediate representations, provides a testbench for modifying and adapting old, inventing new, and also integrating compilation techniques. The evaluation of the experiments is aided by the opportunity to take various performance-related measurements at many different levels of abstraction. The modular design also makes it possible to estimate the effort needed to implement each component and the difficulty of integrating two or more components. We expect some of the techniques introduced with this project to be further developed independently and eventually find their place also in other compilers and program transformers.

1.3 Summary

This dissertation is organized as follows:

- Chapter 2 describes the generalized intensional transformation, which transforms programs written in a first-order language into programs in a zero-order language with intensional operators.
- Chapter 3 describes a technique for using the well-known method of defunctionalization in a separate-compilation setting.
- Chapter 4 describes the design of the compiler's LAR back-end (currently generating C code).
- Chapter 5 presents the results of an early attempt involving the generation of C code that uses labels and a custom stack instead of C functions.
- Chapter 6 discusses the issue of garbage collection and explains the reasons for choosing the Boehm-Demers-Weiser garbage collector for integration in our implementation.
- Chapter 7 describes the implementation of two optimizations in the LAR back-end of our compiler, targeting at better runtime performance and an improved memory footprint.
- Chapter 8 examines the usefulness of lambda lifting in our case and describes the design of the lambda lifter developed for our compiler.
- Chapter 9 summarizes the contributions of this dissertation and concludes with some ideas and directions for the further development of the project.

Figure 1.1 provides a simple visualization of GIC's structure which should be useful for quickly becoming familiar with the compiler's design and should also help someone studying a part of this dissertation to quickly identify the corresponding level in the compilation chain.

1.4 Styling and coloring conventions

For the reader's convenience, some styling and coloring conventions have been used throughout this dissertation. Most of them are the usual ones and their purpose is obvious. However, in this work we describe several program transformations, and most of the times we provide helpful accompaning examples. In these cases, we use a light grey background color for the input and output programs to make them distinguishable and make easier the comparison between them, while we use no background color for all other code snippets (such as code that is part of GIC's implementation, code that is used to demonstrate intermediate steps of some transformation or the syntax of a language, etc.).

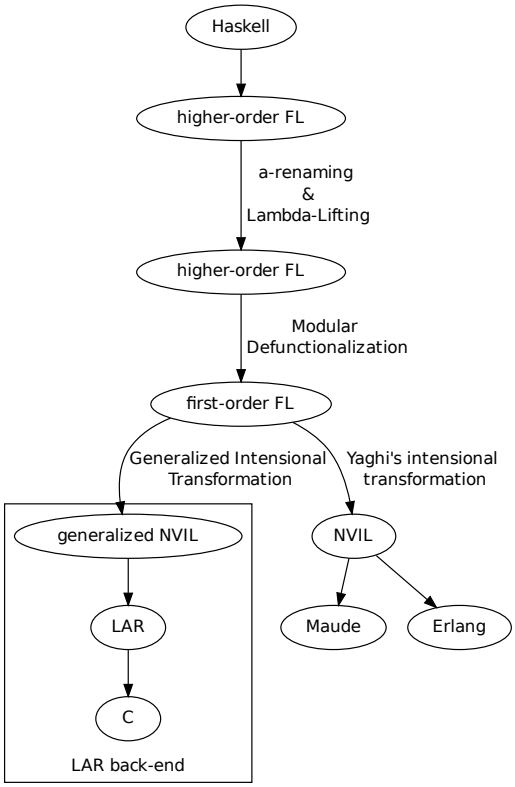

Figure 1.1: Overview of the compiler's structure.

Chapter 2

The intensional transformation

2.1 Introduction

The following two subsections are a high-level overview of the ideas behind intensional programming languages and a short review on the history of transformations of functional programs to equivalent intensional ones. This is not intended to be an introduction to intensional programming; the focus here is on using intensional languages as a back-end in the compilation process of lazy functional programming languages. In Section 2.2 a transformation from a first-order lazy functional language with user-defined data types to a zero-order intensional language with user-defined data types is presented in detail.

2.1.1 Intensional programming

The intensional programming paradigm is inspired from intensional logic [Mont70]. Intensional languages contain context-switching operators, which can be understood operationally as manipulators of hidden parameters. Intensions (values) in an intensional language vary over the space of these hidden parameters. One such language is Lucid [Ashc77]: the value of a Lucid expression depends on a hidden time parameter and the language's intensional operators (`first`, `next`, `fby`) move us between time points.

In the case of Lucid, the semantics of the operators (intensional and conventional) can be formalized by the following equations:

$$
\begin{aligned}
(x + y)_t &= x_t + y_t \\
(\texttt{first}\ x)_t &= x_0 \\
(\texttt{next}\ x)_t &= x_{t+1} \\
(x\ \texttt{fby}\ y)_t &= \begin{cases} x_0 & \text{if } t = 0 \\ y_{t-1} & \text{if } t > 0 \end{cases}
\end{aligned}
$$

The `fby` operator can express iteration: the first argument is the initial value and the second describes how to derive each succeeding value. The following Lucid program computes the stream $\langle 1, 1, 2, 3, 5, \ldots \rangle$ of all Fibonacci numbers:

```
result = fib
fib    = 0 fby (1 fby (fib + next fib))
```

Observe that, in Lucid, assignment statements are actually equations and the order of statements is therefore irrelevant. Lucid programs are usually evaluated using a computational model called *eduction* [Ashc85, Faus87]. An eductive computation propagates demands for the values of specific variables at

specific contexts. A demand for a variable in some context is converted into a demand for its defining expression in the same context which, in turn, generates demands for its subexpressions (including the variables occurring in them).

In Subsection 2.2 the eduction computation model is presented for another intensional language (NVIL), which is also used as an intermediate representation in GIC, and therefore we do not provide any details on eduction here.

2.1.2 From functional to intensional

A transformation from first-order functional programs to intensional programs of nullary variables was for the first time presented in Yaghi's PhD dissertation [Yagh84]. Programs in Yaghi's intensional language (Nullary Variables Intensional Language – NVIL) can be evaluated using *eduction*, which is a demand-driven tagged dataflow model. This makes the transformation suitable for implementing first-order functional programs on dataflow architectures [Arvi90] in a straightforward manner. The demand-driven execution model of the target language corresponds to the call-by-name evaluation strategy for the source language.

Yaghi's intensional language only supports nullary variable definitions. The main idea behind his work is that (first-order) functions (and their formal parameters) can be understood as values varying over the space of invocations (calls). The extension of a formal parameter at one of these is the appropriate actual parameter. The language has two intensional operators: `call` and `actuals`.

The following NVIL program computes the 4^{th} Fibonacci number:

```
result = call0(fib)
fib    = if (n<2) then 1 else call1(fib) + call2(fib)
n      = actuals(4, n-1, n-2)
```

Notice that while Lucid intensions where actually streams of simple data (values varying over a time parameter), here we have tree-shaped intensions (see Figure 2.1). For example, the definition of `n` in terms of `actuals` expresses the fact that intension `n` is a tree with its root labeled with 4, the root of the left subtree is equal to the current root minus 1 and the root of the right subtree is equal to the current root minus 2. This recursively defines a tree—from now on, "tree" will be used as a synonym of "intension". Now, `fib` operates on "tree" (intension) `n` and constructs another tree, whose root is labeled with the 4^{th} Fibonacci number. The definition of `fib` can be understood as follows: the value of a node of the `fib` tree is equal to 1 if the value of the corresponding node of the `n` tree is less than 2; otherwise it is equal to the sum of the values found at the roots of the left and right subtree of the node. Now it is easy to see how the intensional `call` operator changes the context in this example: $\texttt{call}_0\,\texttt{(fib)}$ returns the root of the `fib` tree, $\texttt{call}_1\,\texttt{(fib)}$ selects the root of the left subtree of the current node of `fib` and $\texttt{call}_2\,\texttt{(fib)}$ selects the root of the right subtree of the current node of `fib`.

Given intensions $a_0, \ldots, a_n$, the semantics of the (intensional and conventional) operators can be formalized by the following equations:

$$\begin{aligned}
(\texttt{call}_i\,(a))(w) &= a(i:w)\\
(\texttt{actuals}\,(a_0,\ldots,a_{n-1}))(i:w) &= a_i(w)\\
(\texttt{c}(a_0,\ldots,a_{n-1}))(w) &= c\,(a_0(w),\ldots,e_{n-1}(w))
\end{aligned}$$

From a context/"hidden parameters" perspective, intuitively, $\texttt{call}_i$ augments the context w (which is a list) by prefixing it with i. The `actuals` operator takes the head i of a list (representing the context)

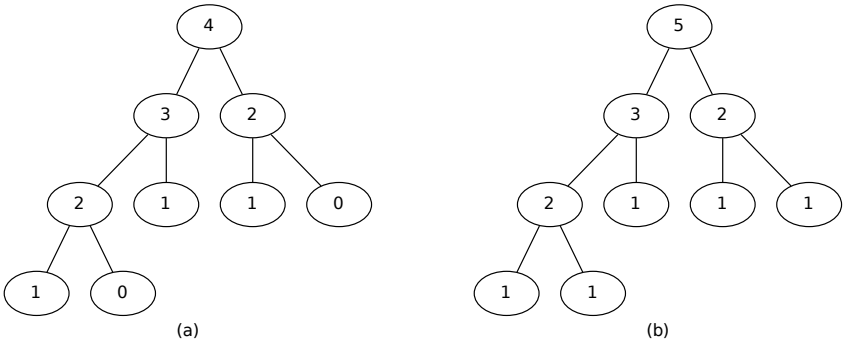

Figure 2.1: Intension `n` (a) and intension `fib` (b)

$$
\begin{array}{lcl}
\mathit{EVAL}_p(v, w) & = & \mathit{EVAL}_p(\mathit{body}(v, p), w) \\
\mathit{EVAL}_p(\texttt{call}_{\,i}\,(e), w) & = & \mathit{EVAL}_p(e, i : w) \\
\mathit{EVAL}_p(\texttt{actuals}\,(e_0, \ldots, e_{n-1}), i : w) & = & \mathit{EVAL}_p(e_i, w) \\
\mathit{EVAL}_p(\texttt{c}(e_0, \ldots, e_{n-1}), w) & = & c(\mathit{EVAL}_p(e_0, w), \ldots, \mathit{EVAL}_p(e_{n-1}, w))
\end{array}
$$

Figure 2.2: The *EVAL* function for the intensional language.

and uses it to select its i^{th} argument. The n-ary constant `c` represents all usual constructs of functional languages (like nullary constants, if-then-else, arithmetic/boolean operators, etc.).

As in the case of Lucid, in the eductive execution model a demand for a variable is converted into a demand for its defining expression. This fact together with the semantic equations for the operators allow us to directly derive an eductive evaluator function for NVIL presented in Figure 2.2. Notice that EVAL_p is parametrized by the program p, which is to be evaluated, and the function $\mathit{body}(v, p)$ returns the defining expression of a variable v in program p.

Having got a feeling of the target intensional language we can summarize the steps involved in Yaghi's transformation:

1. For every function `f` defined in the source functional program, enumerate the textual occurrences of calls to `f`, including calls in the body of the definition of `f`, starting at 0.
2. Replace the i^{th} call of `f` in the source program with $\texttt{call}_{\,i}\,\texttt{(f)}$ and remove the formal parameters from the definition of `f`.
3. Introduce a new definition for each formal parameter of `f`. The right hand side of each such definition is the operator `actuals` applied to the list of the actual parameters corresponding to the formal parameter in question, sorted in the order in which the call sites of `f` are enumerated.

For the input first-order functional program the following assumptions are adopted:

- A distinguished nullary variable `result` is defined, which does not appear in the body of any definition and the value of which is considered as the result of the evaluation of the program.
- Every variable name is defined or appears as a function's formal parameter at most once in the whole program—i.e. all variable names are distinct. This restriction can always be satisfied by α-renaming.

$$
\begin{aligned}
& EVAL(\texttt{result}, [\,]) \\
= \;& EVAL(\texttt{call}_0\texttt{(f) + call}_1\texttt{(f)}, [\,]) \\
= \;& EVAL(\texttt{call}_0\texttt{(f)}, [\,]) + EVAL(\texttt{call}_1\texttt{(f)}, [\,]) \\
= \;& EVAL(\texttt{f}, [0]) + EVAL(\texttt{f}, [1]) \\
= \;& EVAL(\texttt{call}_0\texttt{(g)}, [0]) + EVAL(\texttt{y}, [0]) + EVAL(\texttt{call}_0\texttt{(g)}, [1]) + EVAL(\texttt{y}, [1]) \\
= \;& EVAL(\texttt{g}, [0,0]) + EVAL(\texttt{y}, [0]) + EVAL(\texttt{g}, [0,1]) + EVAL(\texttt{y}, [1]) \\
= \;& EVAL(\texttt{z}, [0,0]) + EVAL(\texttt{y}, [0]) + EVAL(\texttt{z}, [0,1]) + EVAL(\texttt{y}, [1]) \\
= \;& EVAL(\texttt{actuals (x+1)}, [0,0]) + EVAL(\texttt{actuals (6,9)}, [0]) + \\
& EVAL(\texttt{actuals (x+1)}, [0,1]) + EVAL(\texttt{actuals (6,9)}, [1]) + \\
= \;& EVAL(\texttt{x+1}, [0]) + EVAL(\texttt{6}, [\,]) + EVAL(\texttt{x+1}, [1]) + EVAL(\texttt{9}, [\,]) \\
= \;& EVAL(\texttt{actuals (4,5)}, [0]) + EVAL(\texttt{1}, [0]) + 6 + EVAL(\texttt{actuals (4,5)}, [1]) + EVAL(\texttt{1}, [1]) + 9 \\
= \;& 4 + 1 + 6 + 5 + 1 + 9 \\
= \;& 26
\end{aligned}
$$

Figure 2.3: Execution of the target intensional program.

- The formal parameters of a function definition can only appear in the definition's body and the only variable names that can appear in a program are those defined in it and their formal parameters. Complying with this restriction can be taken for granted for a well-typed functional program.

Applying this algorithm on the following functional program, which computes the 4^{th} Fibonacci number recursively,

```
result = fib(4)
fib(n) = if (n<2) then 1 else fib(n-1) + fib(n-2)
```

transforms it exactly to the intensional program presented earlier as an example.

Applying the transformation on the following functional program:

```
result  = f(4,6) + f(5,9)
f(x,y)  = g(x+1) + y
g(z)    = z
```

yields the intensional program:

```
result  = call0(f) + call1(f)
f       = call0(g) + y
g       = z
x       = actuals(4,5)
y       = actuals(6,9)
z       = actuals(x+1)
```

As an example, we can use our eductive evaluator to ask for the value of variable `result` in our program. A trace of the execution is shown in Figure 2.3.

Rondogiannis and Wadge gave a precise formulation of the transformation and a proof of its correctness [Rond97]. They also extended the transformation in order to handle programs written in a higher-order functional language [Rond99]. However, in their higher-order source language function names can be passed as parameters but functions cannot be returned as results (i.e. "currying" is effectively forbidden) and operators are first-order.

Towards the Practical Application of the Intensional Transformation

Regardless of the initial motivation for designing intensional languages, the possibility to transform lazy functional programs into equivalent intensional ones also revealed a new compilation route for lazy functional languages, with some kind of eductive evaluator as a back-end. Eduction is a simple computational model which can potentially achieve high execution speed. Faustini and Wadge have described a way to efficiently implement eduction for Lucid [Faus87]. This involves the memoization of computed values in an associative memory ("warehouse") in order to avoid unnecessary recomputation and a heuristic algorithm for periodically clearing out the warehouse in order to avoid very high memory consumption. The demand-driven nature of eduction, also supported by an associative memory as described above, matches with call-by-need operational semantics for the source language. Therefore, this new compilation route for lazy functional languages also provides a novel way for handling laziness: call-by-name semantics are elegantly taken care of by the intensional transformation and laziness is handled only by the low-level intensional back-end.

The above observations suggest that there are two important things to consider before constructing a real functional language compiler that realizes the novel ideas:

- We would like to be able to compile a full-blown lazy functional language such as Haskell without excluding usual features such as higher-order functions, currying, and data types.
- Although efficient interpretation may be useful, in the case of a language such as Haskell the primary target is usually compilation to machine code. Therefore, given a conventional target machine, the dataflow computational model of eduction must somehow become control-flow in order to avoid interpretation.

Possible benefits from such an approach to the compilation of functional languages include:

- A new, potentially both elegant and efficient way to handle laziness.
- An additional intermediate (intensional) language in the compilation chain suitable for formal reasoning about program properties/correctness—after all, this was one of the initial motivations behind the effort for designing and implementing intensional languages such as Lucid.

In fact, Faustini and Wadge's approach [Faus87] seems to already be close enough to machine code generation. Grivas' implementation of Yaghi's intensional language uses a garbage-collected warehouse and translates the equations of the intensional program into C functions taking a "world" as an argument and returning a (simple) value as a result [Griv04]. This C back-end in Grivas' work is combined with a front-end implementing the higher-order intensional transformation [Rond99] in order to derive a zero-order intensional program from a higher-order functional program. The comparison of this scheme with popular lazy functional language compilers and interpreters in terms of the execution time of certain benchmarks yielded encouraging but inconclusive results.

In another attempt to compile intensional programs to efficient low-level code, Charalambidis, Grivas, Papaspyrou, and Rondogiannis abandoned the idea of a warehouse and used *lazy activation records* instead, i.e. activation records in which some entries are filled on-demand [Char08]. Their approach, extending the scheme proposed in [Rond94], presents many similarities to the traditional use of activation records, which are used to hold a function's actual parameters and are organized as a stack in memory, with two major differences: actual parameters are filled in on demand and the construction and destruction of activation records is indicated by the intensional operators in the zero-order program, and is actually controlled by their low-level counterparts in the final program.

$$
\begin{array}{lll}
p & ::= & d_0,\ \ldots,\ d_n & \textit{program} \\
d & ::= & f(v_0,\ \ldots,\ v_{n-1}) = e & \textit{definition} \\
e & ::= & c(e_0,\ \ldots,\ e_{n-1}) \mid f(e_0,\ \ldots,\ e_{n-1}) \mid \kappa(e_0,\ \ldots,\ e_{n-1}) & \textit{expression} \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ \{\ b_0\ \texttt{;}\ \ldots\ \texttt{;}\ b_n\ \} \mid \#^m(v) & \\
b & ::= & \kappa(v_0,\ \ldots,\ v_{n-1}) \rightarrow e & \textit{case clause}
\end{array}
$$

Figure 2.4: FOL Syntax

Stacks of activation records are represented by linked lists. Tags are sets of pointers pointing at the first elements of activation record stacks. A bit is used to distinguish between value and name arguments in the record and the representation of the argument (a name or a value) follows. A variable identifier (name) is represented by a pointer to the code that implements the corresponding definition. A simplified version of lazy activation records, which is also used in GIC, is described in detail in Section 4.1, so we will not elaborate on this design here.

Their implementation also uses the higher-order intensional transformation (with minor modifications) as a front-end. The intensional language used here is a slightly modified version of Yaghi's language, which stores additional information in its syntax that is used to improve the performance of the execution model.

The approaches described above, although successful in many aspects, can only handle a restricted higher-order language where functions cannot be returned as results and all operators are first-order. This is because that is exactly the language that the higher-order intensional transformation can handle.

2.2 The generalized intensional transformation

In order to support a higher-order functional source language with user-defined data types that provides all the usual features without restrictions, Fourtounis, Papaspyrou, and Rondogiannis use defunctionalization for obtaining a first-order functional program with user-defined data types from an arbitrary higher-order one, and modify the first-order intensional transformation to handle the user-defined data types in the source language and to target an extended variant of NVIL that supports user-defined data types [Four11, Four13b].

Defunctionalization reduces the two aforementioned problems into one, namely the handling of user-defined data types (and pattern matching) under the (first-order) intensional transformation. In order to be able to apply the intensional transformation on the program resulting from defunctionalization all constructors are wrapped in functions, all occurrences of constructors are replaced by calls to their wrappers, and the pattern-bound variable names are kept the same with the names of the formals of the corresponding wrapper. Further explanations are given later.

2.2.1 The source language FOL

The first-order functional language FOL serves as the source language of the generalized intensional transformation. Essentially it is a typed version of Yaghi's source language, with call-by-need semantics, which also supports user-defined data types. Its syntax is presented in Figure 2.4, where f and v range over variables, c ranges over constants, and κ ranges over constructors. Notice that FOL's syntax essentially matches the syntax of DHL (3.2.2), which is the target language of the Modular Defunctionalization Transformation described in Subsection 3.2.3.

$$
\begin{array}{lll}
p & ::= & d_0,\ \ldots,\ d_n & \textit{program} \\
d & ::= & f = e & \textit{definition} \\
e & ::= & c(e_0,\ \ldots,\ e_{n-1}) \mid f \mid \kappa \mid \texttt{case}\ e\ \texttt{of}\ \{\ b_0\ \texttt{;}\ \ldots\ \texttt{;}\ b_n\ \} & \textit{expression} \\
 & \mid & \#^m(e) \mid \texttt{call}_\ell(e) \mid \texttt{actuals}\,(\langle e_\ell \rangle_{\ell \in I}) & \\
b & ::= & \kappa \to e & \textit{case clause}
\end{array}
$$

Figure 2.5: NVIL Syntax

The assumptions made for Yaghi's functional language (2.1.2) also apply here. Furthermore, as outlined earlier, for each constructor κ with n arguments a wrapper function is introduced, defined as follows:

$$f_\kappa(v_0, \ldots, v_{n-1}) = \kappa(v_0, \ldots, v_{n-1})$$

and all occurrences of κ in the program will be replaced by occurrences of f_κ. In all `case` expressions, patterns that match the constructor κ will use (as pattern-bound variables) the same variables $v_0, \ldots, v_{n-1}$ that appear in the definition of f_κ. But this cannot be achieved by simple renaming in the case of nested `case` expressions. In order to resolve this issue, a special form of expressions $\#^m(v)$ is introduced. Intuitively, $\#^m(v)$ corresponds to the variable v that is bound in a pattern of the m-th enclosing `case` expression. The idea is illustrated in the following example, where the expression in the left (in Haskell syntax) is transformed to FOL code that meets the aforementioned requirement:

```
case l of
  Nil → 0
  Cons x xs →
    case xs of
      Nil → x
      Cons y ys → x+y
```

```
case l of
  Nil → 0
  Cons (h, t) →
    case #0(t) of
      Nil → #1(h)
      Cons (h, t) → #1(h) + #0(h)
```

Here the same set of variables (h, t) is used in both patterns for *Cons*; `x` and `y`, which both correspond to h, are distinguished by the value of m (the nesting depth of `case` expressions).

2.2.2 The target language NVIL

NVIL, informally described in Subsection 2.1.2, is extended to support user-defined data types. The extended syntax is presented in Figure 2.5 There are two additional refinements:

- The syntax of the intensional operators `call` and `actuals` is slightly changed. Instead of being labeled by a number i `call` is now labeled by an element l of the set *Labels*. Respectively, `actuals` accepts a sequence of expressions e_l indexed by labels ranging over $I \subseteq$ *Labels*. This convention is useful for the definition of the intensional transformation and it does not affect the semantics of NVIL: again, `call` adds a new label to the context and `actuals` selects the expression to evaluate based on the current label, which is also removed from the context.
- $\#^m(v)$ is replaced by the more general $\#^m(e)$. Intuitively, this expression's semantics in an arbitrary context is the semantics of e in the context corresponding to the m-th enclosing `case` expression—more explanations on these contexts are provided below.

The semantics of NVIL is presented in Figure 2.6 in the form of an evaluation function $EVAL_p(e, w)$, where p is the program, e is the expression to be evaluated, and w is the intensional context.

In contrast to the simple structure of contexts (lists of labels) of Yaghi's intensional language, the introduction of user-defined data types requires a more complex kind of contexts. Contexts are defined by the following grammar.

$$
\begin{array}{l}
\mathit{EVAL}_p(c(e_0,\ \ldots,\ e_{n-1}), w) \ = \ c(\mathit{EVAL}_p(e_0, w), \ldots, \mathit{EVAL}_p(e_{n-1}, w)) \\
\mathit{EVAL}_p(f, w) \ = \ \mathit{EVAL}_p(\mathit{body}(f, p), w) \\
\mathit{EVAL}_p(\kappa, w) \ = \ \langle \kappa, w \rangle \\
\mathit{EVAL}_p(\texttt{case}\ e\ \texttt{of}\ \{\kappa_0 \to e_0;\ \ldots;\ \kappa_n \to e_n\}, \langle \ell, w, \mu \rangle) \ = \ \mathit{EVAL}_p(e_i, \langle \ell, w, w' : \mu \rangle) \\
\quad \text{if } \mathit{EVAL}_p(e, \langle \ell, w, \mu \rangle) = \langle \kappa_i, w' \rangle \\
\mathit{EVAL}_p(\#^m(e), \langle \ell, w, \mu \rangle) \ = \ \mathit{EVAL}_p(e, \mu_m) \\
\mathit{EVAL}_p(\texttt{call}_\ell(e), w) \ = \ \mathit{EVAL}_p(e, \langle \ell, w, \bullet \rangle) \\
\mathit{EVAL}_p(\texttt{actuals}\,(\langle e_\ell \rangle_{\ell \in I}), \langle \ell, w, \mu \rangle) \ = \ \mathit{EVAL}_p(e_\ell, w)
\end{array}
$$

Figure 2.6: NVIL Semantics

$$
\begin{array}{rcl}
w & ::= & \bullet \mid \langle \ell, w, \mu \rangle \\
\mu & ::= & \bullet \mid w : \mu
\end{array}
$$

The new element is μ, which is a list of contexts corresponding to nested `case` expressions, whereas w roughly corresponds to the familiar notion of context described and used in Subsection 2.1.2: it has the form of a linked list holding elements of the set *Labels* instead of numbers; however, every node of this list also contains a μ element.

The result of function $\mathit{EVAL}_p(e, w)$ is either a ground value, which is returned by the meaning of some operator c (e.g., an integer number), or a pair of the form $\langle \kappa, w \rangle$, which corresponds to a value of a user-defined data type—note that these pairs belong to the meta-level. In the latter case, κ is the constructor that was used to build this value and w is the context that must be used to evaluate the constructor's arguments. This semantics is captured in the equation for $\mathit{EVAL}_p(\kappa, w)$; remember that such expressions can only occur in the bodies of functions f_κ that have been introduced for all constructors κ.

The semantic equations for the new syntactic constructs of NVIL (`case` and $\#^m(e)$) can be understood as follows:

- In a `case` expression first e is evaluated and is found to be of the form $\langle \kappa_i, w' \rangle$ for some constructor κ_i that is mentioned in one of the clauses of `case`. Then the body e_i of that clause is evaluated with context w' prepended to the list μ of contexts corresponding to nested `case` expressions.
- An expression $\#^m(e)$ is evaluated by evaluating e in the context μ_m, which is the context in the m-th position of the list μ. More specifically, if the expression e_i mentioned in the previous case uses the arguments of the constructor κ then it will contain an expression $\#^m(e)$ that evaluates them in the appropriate context.

2.2.3 The transformation

The transformation described in this subsection (generalized intensional transformation) transforms FOL programs into semantically equivalent NVIL programs. The transformation is formally defined in Figure 2.7. A high-level description of the transformation's basic operations is given below:

- Function $\mathit{Trans}(p)$ removes the formal parameters from all definitions and adds one extra definition for every formal parameter of every function in program p.
- Given a function f with formal parameters $v_0, \ldots, v_{n-1}$, the function $\mathit{actdefs}(f, p)$ creates one `actuals` definition for each v_j; this definition contains a sequence of all the (processed) actual parameters of f in p that correspond to the j-th position.

$$
\begin{array}{lcl}
\mathcal{E}(c(e_0,\ldots,e_{n-1})) & = & c(\mathcal{E}(e_0),\ldots,\mathcal{E}(e_{n-1})) \\
\mathcal{E}(f) & = & f \\
\mathcal{E}(f(e_0,\ldots,e_n)) & = & \texttt{call}_\ell(f) \quad \text{where } \ell = \langle e_0,\ldots,e_n\rangle \\
\mathcal{E}(\kappa(e_0,\ldots,e_{n-1})) & = & \kappa \\
\mathcal{E}(\texttt{case } e \texttt{ of } \{b_0\texttt{;}\ldots\texttt{;}b_n\}) & = & \texttt{case } \mathcal{E}(e) \texttt{ of } \{\mathcal{B}(b_0);\ \ldots;\ \mathcal{B}(b_n)\texttt{}\} \\
\mathcal{E}(\#^m(e)) & = & \#^m(\mathcal{E}(e)) \\
\mathcal{B}(\kappa(v_0,\ \ldots,\ v_{n-1}) \to e) & = & \kappa \to \mathcal{E}(e)
\end{array}
$$

$$
\mathit{labels}(f,p) = \{\langle e_0,\ldots,e_{n-1}\rangle \mid f(e_0,\ldots,e_{n-1}) \text{ in } p\}
$$

$$
\mathit{actdefs}(f,p) = \bigcup_{j=0}^{n-1} \{v_j = \texttt{actuals}\,(\langle\mathcal{E}(l_j)\rangle_{l\in I})\}
$$

where $v_0,\ldots,v_{n-1}$ are the formal parameters of f and $I = \mathit{labels}(f,p)$

$$
\mathit{Trans}(p) = \bigcup_{f(v_0,\ldots,v_{n-1})=e \text{ in } p} \{f = \mathcal{E}(e)\} \cup \mathit{actdefs}(f,p)
$$

Figure 2.7: The transformation algorithm from FOFL to NVIL.

- The primary role of functions $\mathcal{E}$ and $\mathcal{B}$ is to replace function calls with the corresponding applications of of the `call` operator.
- $\mathit{labels}(f,p)$ is the set of labels of the calls to function f in program p, which will form the indices of the `call` operators in the NVIL program. The label of a function call $f(e_0,\ldots,e_{n-1})$ is defined to be the sequence of its arguments $\langle e_0,\ldots,e_{n-1}\rangle$. If we consider labels as sequences it makes sense (at least as a notation) to index them with numbers: l_j in Figure 2.7 indicates the j-th element of the sequence, i.e. the j-th actual parameter of the call that is marked with label l.

Observe that, in contrast with Yaghi's intensional transformation, syntactically identical function calls in the source program receive exactly the same label under this scheme.

Applying the generalized intensional transformation on the following functional program:

```
result  = f(4,6) + f(5,9)
f(x,y)  = g(x+1) + y
g(z)    = z
```

yields the intensional program:

```
result  = call⟨4,6⟩(f) + call⟨5,9⟩(f)
f       = call⟨x+1⟩(g) + y
g       = z
x       = actuals(4⟨4,6⟩,5⟨5,9⟩)
y       = actuals(6⟨4,6⟩,9⟨5,9⟩)
z       = actuals(x+1⟨x+1⟩)
```

Now we can justify the need for the constructors-as-functions approach and for keeping pattern-bound variable names the same as the names of the formals of the corresponding wrapper function (both described in Subsection 2.2.1): intuitively, as constructors lack named formal parameters, wrapper functions is a way to provide such named formal parameters and expose them to the intensional transformation. Otherwise there would be no way to access in the NVIL program (where constructors have no arguments at all) the intensions corresponding to a constructor's actual parameters (in the FOL program). Keeping pattern-bound variable names the same as the formals of a constructor's wrapper

function makes it possible to refer to these intensions in the body of a `case` clause and the syntactic form $\#^m(e)$ provides the right context to the `actuals` operator that will return the value corresponding to the constructor's actual parameter in the FOL program.

Chapter 3

Separate compilation with defunctionalization

In this chapter we provide a self-contained presentation of the Modular Defunctionalization technique, which makes it possible to use defunctionalization in a compiler that supports separate compilation to native code, and which is currently a part of GIC's compilation chain. Section 3.1 discusses defunctionalization with emphasis on its utilization as a compilation technique, and provides some examples. Section 3.2 describes Modular Defunctionalization in detail; however, the details are only given for the sake of completeness here, as they are not necessary for understanding the material provided in the following chapters of this dissertation—perhaps with the exception of the description of the information collected for every module during separate compilation.

3.1 Defunctionalization

In this section we review defunctionalization, provide some examples of its application and argue that it is a plausible compilation technique that solves the problem of closure representation in the case of higher-order functional languages. Finally, the integration of defunctionalization into GIC is briefly discussed in order to pave the path for the detailed description of the Modular Defunctionalization Transformation in the following section.

3.1.1 Introduction

Defunctionalization is a transformation invented by Reynolds [Reyn72] that transforms higher-order functional programs to first-order programs. This is done by encoding higher-order values as first-order data and realizing applications originally involving higher-order values as applications of dispatching functions.

Reynolds originally used his technique to investigate the nature of higher-order functions. He derives an interpreter for a higher-order language in a first-order language and he shows that the defunctionalized version of a continuation passing style interpreter for a higher-order language is very similar to Landin's SECD machine [Reyn72], effectively providing an association between mathematical and machine-like definitions of higher-order language semantics. Reynolds also observes however that, regardless of the initial motivation, defunctionalization is applicable to any higher-order functional program.

Since then, defunctionalization has been used in many different contexts; for example, Danvy and Nielsen selectively defunctionalize the continuations of CPS-converted programs to derive a version that uses first-order accumulators and also show that defunctionalizing a function that uses a higher-order recursive auxiliary function yields a first-order version with an accumulator [Danv01]. However, we will not elaborate on such issues; instead we will focus on the use of defunctionalization as a basic

compilation technique which transforms a higher-order program (as a whole) to an equivalent first-order program, effectively bringing us a step closer to code generation.

Bell, Bellegarde, and Hook presented formally a defunctionalizing transformation that preserves types in the case of ML-polymorphism (let-polymorphism) [Bell97]. A simpler type-safe transformation is easily derived in the case of a simply-typed source language—this forms the basis of the variant of defunctionalization currently used by GIC, as we will see later in this chapter.

3.1.2 Examples

Some examples involving higher-order functions are presented below along with their defunctionalized counterparts. The examples are written in a Haskell-like language with simple types and the defunctionalization variant used here is derived as a special case of the approach of Bell *et al.* —actually it is very close to the one GIC currently uses.

For a first example we take the following short program:

```
result   = high (add 1) 1 + high inc 2
high g x = g x
inc z    = z + 1
add a b  = a + b
```

We can see that there are two higher-order values that are passed as arguments and/or returned as results:

```
add 1 :: Int → Int    -- \b -> 1 + b
inc   :: Int → Int
```

These will be *represented* by first-order data, and their application will be performed by a special *apply* function. Observe that these values have the same type, therefore only one data type (with one constructor for each higher-order value) and one *apply* function will be defined:

```
data Closure = Add Int | Inc

apply clos arg =
  case clos of
    Add i → add i arg
    Inc   → inc arg
```

Now we have to consider where are the higher-order expressions mentioned above applied in the original program. We see that this only happens in the body of `high`:

```
high g x  = apply g x
```

Putting it all together we get an (extensionally) equivalent program that uses only first-order functions:

```
data Clos = Add Int | Inc

apply clos arg =
  case clos of
    Add i → add i arg
    Inc   → inc arg

result    = high (Add 1) 1 + high Inc 2
high g x  = apply g x
inc z     = z + 1
add a b   = a + b
```

Let's examine another example where higher-order values of different types are involved:

```
result    = high1 add 1 1 + high2 inc 2
high1 g x = g x
high2 g x = g x
inc z     = z + 1
add a b   = a + b
```

There are two higher-order values that are passed as arguments:

```
add :: Int → Int → Int
inc :: Int → Int
```

and one that is returned as result by a full application of `high1`:

```
add (..) :: Int → Int   -- partial application of add
```

These will be represented by first-order data:

```
-- Closures with residual type Int -> Int -> Int
data ClosIII = Add1

-- Closures with residual type Int -> Int
data ClosII  = Add2 Int | Inc

-- Fully apply closures with residual
-- type Int -> Int -> Int
apply_III clos arg1 arg2 =
  case clos of
    Add1   → add arg1 arg2

-- Partially apply closures with residual
-- type Int -> Int -> Int
-- on one argument
apply_III_I clos arg =
  case clos of
    Add1   → Add2 arg

-- Fully apply closures with residual
-- type Int -> Int
apply_II clos arg =
  case clos of
    Add2 i → add i arg
    Inc    → inc arg
```

The defunctionalized version of the program is:

```
data ClosIII = Add1
data ClosII  = Add2 Int | Inc

apply_III clos arg1 arg2 =
  case clos of
    Add1   → add arg1 arg2

apply_III_I clos arg =
  case clos of
    Add1   → Add2 arg

apply_II clos arg =
  case clos of
    Add2 i → add i arg
    Inc    → inc arg

result    = apply_II (high1 Add 1) 1 + high2 Inc 2
high1 g x = apply_III_I g x
high2 g x = apply_II g x
inc z     = z + 1
add a b   = a + b
```

3.1.3 Defunctionalizing compilation

Compilers for higher-order functional languages have to deal with the issue of first-class function representation. Danvy and Nielsen summarize some widely used alternatives [Danv01]:

- Functions are pairs (known as *closures*) containing a code pointer and the associated environment, i.e. the values of the variables "occurring free" in the code. This is probably the most common representation of first-class functions in the case of eager functional languages.
- Higher-order programs are defunctionalized into first-order programs. In this case functions are represented as first-order data types which can be thought as pairs containing a tag (indicating the code of the function) and the values of the variables occurring free in this code.
- Transforming functional programs into combinator (i.e. functions without free variables) declarations and then using *graph reduction* also handles first-class functions. This technique is mostly used in the case of lazy functional languages.

Defunctionalization of higher order programs has, in a sense, strong similarities with the closure representation of functions. In the former case, however, facilities of the source language (data types) are used for the representation of functions, whereas in the latter case the representation of functions is usually implemented in a lower-level target language. This makes defunctionalization suitable for a compiler's front-end: it is a meaning- and type-preserving transformation [Bell97, Niel00] which can be used to simplify the design of the compiler's back-end as it allows for a target language that does not provide higher-order functions and therefore is (almost) directly translatable to low-level imperative code.

Defunctionalization in GIC

Defunctionalization and the (variant of the) first-order intensional transformation used by GIC, seem to be an effective combination for compiling a non-strict language: The former makes it possible for the

latter to be applied (avoiding the application of the more complex and restrictive higher-order intensional transformation) whereas the latter provides a simple and elegant way to handle non-strictness. This design greatly simplifies the LAR back-end, which, using the relatively simple technique of Lazy Activation Records and a very minimal runtime environment, manages to produce competitive low-level code. In fact, specifically in the case of the LAR back-end, the runtime overhead imposed by the dispatching functions introduced by defunctionalization is usually eliminated: `case` constructs in FOL (Subsection 2.2.1) are turned into `case` constructs analyzing only the value of a constructor (i.e. a number) in NVIL (Subsection 2.2.2) by the intensional transformation; an optimizing C compiler can usually turn them into jump-tables.

Other compilers that make the same use of defunctionalization (i.e. transform their input to an equivalent first-order program) include MLton [Cejt00] and Boquist's Haskell compiler [Boqu99]. Although defunctionalization is certainly an appealing choice for GIC, to our knowledge it has not been used practically in a separate compilation setting before. As described in the next section, GIC uses a version of defunctionalization adapted to separate compilation.

3.2 Defunctionalization & separate compilation

Defunctionalization has been widely considered as a whole-program transformation. Whole-program transformations are usually unsuitable for realistic compilers because separate compilation is a valuable feature that cannot be spared for good reasons: it makes code reuse, distribution of compiled code as libraries, and tractable recompilation of big code bases possible. Defunctionalization has been used in compilers that run in whole-program mode, such as MLton and UHC, but not in compilers that support separate compilation to native code.

Fourtounis and Papaspyrou introduced a variant of the transformation (*modular defunctionalization*) that supports separate compilation of modules and linking [Four13a]. The transformation currently handles a simply-typed source language and is based on the type-safe variant of defunctionalization of Bell *et al.* [Bell97].

The major problem with "naively" defunctionalizing modules (and subsequently simply linking the object files) is that closure constructors generated by the defunctionalization of a module are only known to dispatching functions generated for the same module. But if higher-order values flow between different modules in the input sources then a dispatcher may suddenly confront an unknown closure constructor. The proposed solution is to collect all closure types, closure constructors and dispatchers from all modules and to postpone code generation for them until link time. All other data types, constructors, and functions can be compiled separately as it is guaranteed that there are no name clashes.

In more detail, this technique applies defunctionalization separately on each module "remembering" the closure constructors that were required and collects this information together with the target code generated for the module. At link time it generates code for the dispatching functions based on the collected information. This makes it in fact a two-stage transformation:

1. *Separate defunctionalization*: Each module is defunctionalized separately. This results to (i) a set of defunctionalized data type declarations; (ii) a set of defunctionalized top-level function definitions; and (iii) information about the closures that were used in this module. The third part serves as the *defunctionalization interface* of the module. At this point, the defunctionalized definitions from each module can be compiled separately to object code, assuming that closure constructors and dispatching functions are external symbols to be resolved later, at link time.

2. *Linking*: The separately defunctionalized code is combined and the missing code for closure constructors and dispatching functions is generated using the defunctionalization interfaces from the

previous step. This code can then be compiled and linked with the rest of the already generated code, to produce the final program.

The following subsections present the source and target languages of the transformation and the Modular Defunctionalization transformation itself. HL_M is a higher-order functional language with data types and modules. HL_M is essentially identical with the FL intermediate language used by GIC. The target language, DHL, is the first-order subset of HL_M without modules. GIC actually defunctionalizes FL to the first-order subset of FL without modules.

3.2.1 The source language HL_M

HL_M is a Haskell-like higher-order functional language with modules. Each HL_M program constitutes of a collection of modules. Each module has:

- a name;
- a list of data types and functions that are imported from other modules;
- a list of data type declarations;
- and a list of function definitions.

The abstract syntax of HL_M is given below, where μ ranges over module names, a over data type names, κ over constructor names, f over constructor names, x over function formals and pattern variables, and b over basic types.

p	$::= m^*$	*program*
m	$::=$ `module` μ `where imports` $I^*\ \delta^*\ d^*$	*module*
I	$::= \mu\ (\mu.a)^*\ (v : \tau)^*$	*import*
δ	$::=$ `data` $\mu.a$ `=` $(\mu.\kappa : \tau)^*$	*data type*
τ	$::= b \mid \mu.a \mid \tau \rightarrow \tau$	*type*
d	$::= \mu.f\ x^*$ `=` e	*definition*
e	$::= (x \mid v \mid op)\ e^* \mid$ `case` e `of` b^*	*expression*
v	$::= \mu.f \mid \mu.\kappa$	*top-level variable*
b	$::= \mu.\kappa\ x^*\ \rightarrow\ e$	*case branch*

Each type name (a), top-level function name (f), and constructor name (κ) are qualified by the name of the module in which they are defined. Function formals and pattern variables (x) are local names, therefore not qualified.

3.2.2 The target language DHL

The target language DHL is the first-order subset of HL_M without modules. DHL is essentially identical with FOL (the source language of the intensional transformation presented in 2.2.1) with only minor differences in the syntax due to the fact that its syntax is directly derived from the syntax of HL_M.

Specifically, the following properties of DHL are directly derived by the general properties of defunctionalization:

- All applications are *full* in terms of function arities.
- All function calls are calls to known functions.
- All functions and data-type constructors are first-order.

Moreover, all module boundaries are eliminated: module qualifiers are considered parts of the names of functions, data types, and constructors; DHL programs are lists of data type declarations and function definitions.

3.2.3 The modular defunctionalization transformation

The two stages of the transformation are formally presented in this subsection. For the rest of the subsection we assume that all type information for the input modules is available and that expressions are annotated with their types when this simplifies the presentation.

Separate defunctionalization stage

The following functions are assumed to produce unique names free of module qualifiers (thus suitable for use in DHL):

- $\mathcal{N}(\mu.a)$, $\mathcal{N}(\mu.f)$, and $\mathcal{N}(\mu.\kappa)$ generate names for module-qualified types, top-level functions, and constructors that appear in the source code of a module;
- $\mathcal{C}\ell(\tau)$ generates the name of a data type corresponding to closures of type τ;
- $\mathcal{C}(v, n)$ generates the name of a constructor corresponding to the closure of v, binding n arguments; and
- $\mathcal{A}(\tau, n)$ generates the name of the closure dispatching function for closures of type τ, supplying n arguments.

Some useful auxiliary functions are defined and shortly described below:

- $\mathsf{arity}(\tau)$ returns the arity of a type (i.e., how many arguments must be supplied before a ground value is reached).

$$\begin{array}{lcl} \mathsf{arity}(b) & \doteq & 0 \\ \mathsf{arity}(\mu.a) & \doteq & 0 \\ \mathsf{arity}(\tau_1 \to \tau_2) & \doteq & 1 + \mathsf{arity}(\tau_2) \end{array}$$

- $|\mathsf{args}(v)|$ returns the number of formal arguments in the definition of v. For any v^τ we always have $|\mathsf{args}(v^\tau)| \leq \mathsf{arity}(\tau)$.
- $\mathsf{ground}(\tau)$ converts higher-order types to ground types, by replacing function types with the corresponding closure types.

$$\begin{array}{lcl} \mathsf{ground}(b) & \doteq & b \\ \mathsf{ground}(\mu.a) & \doteq & \mathcal{N}(\mu.a) \\ \mathsf{ground}(\tau_1 \to \tau_2) & \doteq & \mathcal{C}\ell(\tau_1 \to \tau_2) \end{array}$$

- $\mathsf{lower}(\tau)$ converts higher-order types to first-order, by replacing the arguments of function types with the corresponding closure types, if necessary.

$$
\begin{array}{lcl}
\mathsf{lower}(b) & \doteq & b \\
\mathsf{lower}(\mu.a) & \doteq & \mathcal{N}(\mu.a) \\
\mathsf{lower}(\tau_1 \to \tau_2) & \doteq & \mathsf{ground}(\tau_1) \to \mathsf{lower}(\tau_2)
\end{array}
$$

The defunctionalization transformation includes $\mathcal{T}(\delta)$ for type declarations and $\mathcal{D}(d)$ for top-level function definitions. Transformation $\mathcal{D}(d)$ is defined in terms of $\mathcal{E}(e)$ and $\mathcal{B}(b)$ for expressions and case branches, respectively.

$$
\begin{array}{l}
\mathcal{T}(\texttt{data}\ \mu.a\ =\ \mu.\kappa_1 : \tau_1 \mid \ldots \mid \mu.\kappa_n : \tau_n) \ \doteq\ \texttt{data}\ \mathcal{N}(\mu.a)\ =\ \mathcal{N}(\mu.\kappa_1) : \mathsf{lower}(\tau_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ \mathcal{N}(\mu.\kappa_n) : \mathsf{lower}(\tau_n)
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{D}(\mu.f\ x_1 \ldots x_n\ =\ e) & \doteq & \mathcal{N}(f)\ x_1 \ldots x_n\ =\ \mathcal{E}(e) & \\
\mathcal{E}(x) & \doteq & x & \\
\mathcal{E}(x^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{A}(\tau, n)\ x\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \text{if } n > 0 \\
\mathcal{E}(v^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{N}(v)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \text{if } n = |\mathsf{args}(v)| \\
\mathcal{E}(v^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{C}(v, n)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \text{if } n < |\mathsf{args}(\tau)| \\
\mathcal{E}(v^\tau\ e_1\ \ldots\ e_n) & \doteq & (\mathcal{A}(\tau, n)\ v\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_{|\mathsf{args}(v)|}))\ \mathcal{E}(e_{|\mathsf{args}(v)|+1})\ \ldots\ \mathcal{E}(e_n) & \\
 & & & \text{if } |\mathsf{args}(v)| < n \leq \mathsf{arity}(\tau) \\
\mathcal{E}(op\ e_1\ \ldots\ e_n) & \doteq & op\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \\
\mathcal{E}(\texttt{case}\ e\ \texttt{of}\ b_1\ ;\ \ldots\ ;\ b_n) & \doteq & \texttt{case}\ \mathcal{E}(e)\ \texttt{of}\ \mathcal{B}(b_1)\ ;\ \ldots\ ;\ \mathcal{B}(b_n) & \\
\mathcal{B}(\mu.\kappa\ x_1\ \ldots\ x_n\ \to\ e) & \doteq & \mathcal{N}(\mu.\kappa)\ x_1\ \ldots\ x_n\ \to\ \mathcal{E}(e) &
\end{array}
$$

In principle: (i) partial applications of top-level functions and constructors are replaced by closure constructors; (ii) functional parameters or pattern variables are applied by using the corresponding closure dispatching functions; (iii) data types are also defunctionalized: all higher-order types in the signatures of constructors are replaced by the corresponding closure data types.

During this stage of the transformation we also need to collect information on every possible closure that might be needed based on the given top-level functions and constructors. We define function $\mathcal{F}(v^\tau)$ that returns a set of triples, one for each closure that represents a partial application of the top-level function (or constructor) v:

$$
\begin{array}{lcll}
\mathcal{F}(v^\tau) & \doteq & \mathsf{info}(v, \tau, []) & \\
\mathsf{info}(v, \tau, \tau^*) & \doteq & \{(\tau, \mathcal{N}(v), \tau^*)\} \cup \mathsf{info}(v, \tau_2, \tau^* +\!\!+ [\mathsf{ground}(\tau_1)]) & \text{if } \tau = \tau_1 \to \tau_2 \\
\mathsf{info}(v, \tau, \tau^*) & \doteq & \emptyset & \text{if } \tau \text{ is a ground type}
\end{array}
$$

Each triple contains: (i) the type of the closure; (ii) the name of v; (iii) the types of arguments contained in the closure. As an example, assume the function `add` is defined as follows:

```
add a b c = a + b + c
```

This function has three different kinds of partial applications, that is the case where no argument is applied, the case where one argument is applied, and the case where two arguments are applied:

$$
\begin{array}{l}
\mathcal{F}(\texttt{add}^{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int}}) \ =\ \{\, (\texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int}, \texttt{add}, []), \\
\qquad\qquad\qquad\qquad\quad (\texttt{Int} \to \texttt{Int} \to \texttt{Int}, \texttt{add}, [\texttt{Int}]), \\
\qquad\qquad\qquad\qquad\quad (\texttt{Int} \to \texttt{Int}, \texttt{add}, [\texttt{Int}, \texttt{Int}])\ \}
\end{array}
$$

Linking stage

To link the final program we need to merge all defunctionalized definitions (derived from the separate defunctionalization of modules) and the missing dispatching functions. Let I be the union of closure information from all modules to be linked. For each closure type τ, we generate a definition for $\mathcal{C}\ell(\tau)$ as follows:

$$\texttt{data}\ \mathcal{C}\ell(\tau)\ =\ \{\ \mathcal{C}(x,n) : \tau^* \to \mathcal{C}\ell(\tau)\ |\ (\tau, x, \tau^*) \in I \text{ and } n = \mathsf{arity}(\tau)\ \}$$

As the program is closed at link-time, we only need to create dispatching functions for all constructors in I. For each closure type τ we define the dispatcher for closures representing values of type τ applied on m arguments as follows:

$$\begin{aligned}\mathcal{A}(\tau, m)\ x_0\ x_1\ \ldots\ x_m\ =\ &\texttt{case}\ x_0\ \texttt{of}\\ &\{\ \mathcal{C}(x,n)\ y_1\ \ldots\ y_k \to \mathcal{C}(x, n-m)\ y_1\ \ldots\ y_k\ x_1\ \ldots\ x_m\\ &\quad |\ (\tau, x, \tau^*) \in I \text{ and } n = \mathsf{arity}(\tau) \text{ and } k = |\tau^*|\ \}\end{aligned}$$

Note that defining $\mathcal{C}(x, 0) \doteq x$ allows us to uniformly treat the case of dispatchers returning ground values instead of closures, i.e. dispatchers that apply a closure when all remaining arguments are supplied.

Chapter 4

The LAR back-end

As we can see in the high-level diagram of the compiler (Figure 1.1) presented in Chapter 1, GIC features many back-ends serving different purposes. In this chapter the focus will be solely on the low-level LAR back-end, which aims at the generation of efficient and portable C code.

Section 4.1 presents a method for translating programs in the NVIL language (described in detail in Subsection 2.2.2) to equivalent C programs using lazy activation records (LARs). GIC's LAR back-end is essentially a direct implementation of this method. Some additional low-level details of the back-end are presented in Section 4.2 in order to prepare the reader for the material in Chapters 5, 6 and 7.

4.1 Implementing NVIL with lazy activation records

This section describes a method for translating NVIL programs, that have been derived from the application of the generalized intensional transformation (described in detail in Subsection 2.2.3) on FOL programs, into equivalent C programs. The key idea is to generate a piece of C code for every definition in the intensional program. As NVIL definitions actually define intensions, the C code corresponding to a definition must be parametric on contexts.

In a sense, given an NVIL program p, the resulting C program implements a more efficient version of $EVAL_p$ eductive evaluation function presented in Figure 2.6. Contexts are implemented as LARs, and the rules in Figure 2.6 are implemented with C code parametric on contexts of the form $w = \langle \ell, w', \mu \rangle$.

The runtime system uses a stack and a heap but the only entities that are stored in the stack and the heap are LARs. As explained in Subsection 2.2.2, in the case of NVIL, user-defined data types are (meta-level) pairs containing a constructor and a context in which the constructor's "arguments" must be evaluated. The context-part of a data structure is stored in a LAR representing a context in a μ list whereas the constructor is returned by the C function that computes it on the stack (along with a pointer to the corresponding context). Remember from Subsection 2.2.1 that a constructor's "arguments" are just a set of `actuals` definitions in NVIL corresponding to the constructor's wrapper function's formals in the FOL program and a constructor itself is just a tag (number).

A LAR is created when an expression of the form $\texttt{call}_\ell(f)$ is encountered during the execution of the program. Currently, GIC uses a simple criterion for deciding at each LAR-creation site whether LARs shall be allocated on the stack or on the heap:

- Functions returning ground values (like integers and booleans) or data types with only nullary constructors (i.e. constructors that correspond to nullary constructors in the equivalent FOL program, in which case no $\#^m(e)$ expressions are present in the `case` clauses for this constructor in the NVIL program) allocate their LARs on the stack and therefore deallocate them on return.

- Functions that may return data types built by non-nullary constructors allocate their LARs on the heap.

This scheme allows programs to benefit from the fast stack allocation as long as they do not make extensive use of user-defined data types.

LARs are similar to traditional activation records; however, some of the fields in a LAR are not filled when the LAR is constructed but only when their value is demanded. After that, whenever the value of a function's formal argument is demanded again under the same context during execution, this value can be retrieved directly from the LAR. Effectively, call-by-need semantics are implemented: LARs form the mechanism that handles laziness.

The idea behind the LARs is that a LAR directly corresponds to a context of the form $w = \langle \ell, w', \mu \rangle$ except for the additional fields memoizing values that were described above. More specifically, a LAR contains the following fields:

- *prev*: a pointer to the parent LAR, i.e. the LAR parameter of the function that made the function call that generated this LAR. This directly corresponds to w' above.
- $arg_0, \ldots, arg_{n-1}$: each arg_i is a code pointer which points at the i-th actual parameter of the function call that generated this LAR. These fields correspond to l: they are actually an encoding of l assuming that the labels are sequences of the arguments of function calls (which is exactly the usage of labels the generalized intensional transformation does, as described in Subsection 2.2.3). As these arguments are expressions in an `actuals` definition and therefore intensions, the pointers in the LAR point specifically at `C` functions parametric on the context, i.e. `C` functions receiving a LAR as a parameter.
- $val_0, \ldots, val_{n-1}$: each val_i memoizes the value of the corresponding arg_i. It is initially empty and will be filled on demand: if at some point the code pointed by arg_i is executed and computes a value then this value will be stored in val_i for future use. This implements a call-by-need semantics.
- *nested*: a list of contexts corresponding to nested `case` constructs. In particular, when an expression of the form $\#^m(e)$ is encountered, $nested[m]$ points to the LAR that must be used to evaluate e. Recall from Subsection 2.2.2 that the result of function *EVAL* is either a ground value or a (meta-) pair of the form $\langle \kappa, w'' \rangle$, which corresponds to a value of a user-defined data type. When we have an expression:

  `case` e `of ...`

  the value returned by the evaluation of e is certainly of the latter form. When this expression is evaluated the LAR corresponding to w'' will be stored in the *nested* list. It is evident that *nested* directly corresponds to μ.

4.2 Lazy activation records in GIC

This section describes the implementation of the ideas presented earlier specifically in GIC and the role of the generalized intensional transformationin GIC's compilation chain.

The LAR back-end is fed with first-order programs with user-defined data types (obtained by defunctionalization) in the FL intermediate language and uses the generalized intensional transformation, described in Section 2.2, to transform them into equivalent programs in NVIL. All transformations work at the Abstract Syntax Tree (AST) level.

4.2.1 Overview of the intermediate languages

In order to get a clear picture of the compiler's structure it is probably useful to quickly review the intermediate languages used for the compilation of Haskell to C through the LAR back-end and describe the connections between the intermediate languages used by GIC (see Figure 1.1) and the formal languages introduced in Chapters 2 and 3.

FL

FL is is a typed higher-order lazy functional language with user-defined data types and modules. It corresponds almost directly to HL_M, which was described in Subsection 3.2.1. The differences between them are all minor and only of syntactical nature; the most important being that FL supports local definitions (through the `let .. in ..` construct) and anonymous functions. GIC performs lambda-lifting (described in detail in Chapter 8) before defunctionalization, which eliminates local definitions and anonymous functions and essentially targets the subset of FL exactly corresponding with HL_M.

Defunctionalization in GIC is implemented as an FL-to-FL transformation. However, programs resulting from defunctionalization are guaranteed to belong to the first-order subset of FL without modules—the modular defunctionalization transformation, which is the variant used in GIC, also eliminates boundaries between modules as described in Subsection 3.2.3. This subset of FL corresponds directly to the first-order language FOL, which is the source language of the generalized intensional transformation and is described in Subsection 2.2.1, with only minor differences in syntax.

FL is the intermediate level where some high-level optimizations traditionally applied in the case of functional languages could be performed. However, the interaction of transformations applied at the FL level with the generalized intensional transformation needs further investigation.

ZOIL

ZOIL (Zero Order Intensional Language) exactly corresponds to NVIL described in Subsection 2.2.2—it is just another name for the same thing. Therefore, from now on we will use the name NVIL to also refer to this intermediate representation.

LAR

LAR is an intermediate layer between NVIL (i.e. ZOIL) and C. The code defining its syntax is shown in Figure 4.1. Its purpose is to simplify the C code generation from NVIL code by implementing the idea of "labels as sequences of actual parameters of function calls" described in Subsection 2.2.3. In fact, a LAR program is just a concise description of the corresponding C program. The following discussion about this representation will be kept informal.

The following example may be helpful for visualizing the close connections between NVIL and LAR languages:

Example 4.2.1. Let's use again our familiar sample first-order program:

```
result  = f(4,6) + f(5,9)
f(x,y)  = g(x+1) + y
g(z)    = z
```

We have already seen (Subsection 2.1.2) that Yaghi's intensional transformation yields the intensional program:

```
-- * The LAR language

-- | A LAR program.
data LARProg  = LARProg [Data] [LARBlock]

-- | A function contains a name, a statement, a list of bindings
--   and a list of strict formals.
data LARBlock = Func VName LARStm [VName] [VName]
              | Var VName LARStm

-- | A LAR statement is the body of a definition: it bundles an
--   (optional) actuals operator with a LAR expression.
data LARStm   = LARStm Bool LAExpr

-- | A LAR expression.
data LAExpr   = LACall VName [VName] -- call a variable with a LAR of
    variables
              | LAC CName [LAExpr]  -- built-in constant application
              | ConstrL CstrName    -- constructor call
              | BVL VName BVLoc     -- bound variable (constructor
    projection)
              | CaseL Depth LAExpr [LAPat] -- pattern matching expression

-- | A LAR pattern.
data LAPat    = LAPat CstrName LAExpr Bool
```

Figure 4.1: The LAR language

```
result  = call0(f) + call1(f)
f       = call0(g) + y
g       = z
x       = actuals(4,5)
y       = actuals(6,9)
z       = actuals(x+1)
```

The application of the generalized intensional transformation (Subsection 2.2.3) yields the following intensional program:

```
result  = call⟨4,6⟩(f) + call⟨5,9⟩(f)
f       = call⟨x+1⟩(g) + y
g       = z
x       = actuals(4⟨4,6⟩,5⟨5,9⟩)
y       = actuals(6⟨4,6⟩,9⟨5,9⟩)
z       = actuals(x+1⟨x+1⟩)
```

The equivalent program in the LAR language is:

```
result    = f f_x__0 f_y__0 + f f_x__1 + f_y__1
f f_x f_y = g g_z__0 + f_y
g g_z     = g_z
f_x__0    = ACTUAL.4
f_x__1    = ACTUAL.5
f_y__0    = ACTUAL.6
f_y__1    = ACTUAL.9
g_z__0    = ACTUAL.f_x + 1
```

We can see right away that functions and formal arguments have been re-introduced and that the LAR program presents many similarities with the original *functional* program. However, useful low-level information obtained through performing the intensional transformation is also encoded in the LAR program. All actual parameters are now variables which are explicitly defined through the ACTUAL operator. This is a simplified/restricted form of NVIL's `actuals` operator (or its remains if you prefer, as the LAR program is in fact derived from the NVIL program): ACTUAL . e under the current lazy activation record means to consider e under the activation record pointed by the $prev$ field of the current one.

Now, what is the benefit? It is that we have derived an organization of the initial FOL program into appropriate *pieces of low-level code*: each definition in the LAR program now truly corresponds to a piece of imperative code. The ACTUAL operator provides explicit instructions for "navigating" through the LAR structures during execution and, therefore, laziness is handled properly: the executing C function knows exactly which LAR structure holds a suspended computation and where to save its value.

Although the LAR program seemingly resembles the FOL program, it is instructive to also compare it with the corresponding intensional program, which in fact has more things in common with the LAR program. The first thing to notice is that `actuals` definitions in the NVIL program have been "unfolded" into ACTUAL definitions in the LAR program, effectively eliminating the need for labels. The corresponding functionality of the `actuals` operator, i.e. to choose an expression according to a label, is no longer needed and therefore is not retained by ACTUAL operator. ACTUAL merely switches from the current context to the previous one in the execution, i.e. the equivalent of removing the first label from a list and exposing the next one, which is part of the `actuals` operator's functionality.

Another obvious difference between the NVIL and the LAR program is that in LAR we have function definitions, formal parameters and (conventional) function calls—which is exactly what makes LAR programs look so close to their FOL counterparts. The re-introduction of the notion of function seems to be generally unavoidable in the case of compilation to an imperative language (and, eventually, to machine code for a conventional control-flow architecture), as every piece of code must be parametric on contexts, one way or another. However, this is only a superficial difference: with the convention for labels followed by the generalized intensional transformation (described in Subsection 2.2.3) all the information about the arities of functions and the usage of their parameters is encoded in the labels of the intensional program. The names of the formals in the LAR program are irrelevant—they are just tags for accessing the fields of a context parameter—and in this example the names are chosen specifically for emphasizing the similarity between the LAR and FOL program that was mentioned earlier.

4.2.2 Design summary

Now that the most important features of all intermediate representations have been presented we can summarize the design and functionality of the LAR back-end.

Given a FOL (i.e. a first-order FL) program, what do we need for compiling it into an equivalent imperative program using only lazy activation records?

- We need to place function arguments and expressions to be analyzed in `case` constructs into LARs, so that we handle laziness. The generalized intensional transformation takes care of this: for any function in the FOL program, the NVIL program tells us how the LARs used by this function should look like and adds "abstract code snippets" (`actuals` definitions) to handle these LARs during execution.

- We need an efficient imperative implementation of the context-switching operators. LARs are a form of activation records, which suggests that they should be used as such. Turning `call` expressions into real imperative function calls and eliminating the tags by introducing concrete pieces of imperative code (`ACTUAL` definitions) seems to be a straightforward solution.
- No special care needs to be taken for the user-defined data types after wrapping constructors into functions in the FOL program. Constructor arguments will be kept in LARs just like function arguments—however, these LARs should always be allocated on the heap. A constructor itself is only demanded during the evaluation of a `case` expression and is returned on the stack by the C function that computes it.

Everything seems to be almost in place. Notice that the imperative implementation of FOL programs that has been outlined so far is very direct (the striking similarities between FOL and LAR programs make this evident) although it passes through the intensional transformation. In other words, the FOL program's general structure and appearance are preserved throughout the compilation process.

4.2.3 The C-code generator

Generating C code implementing a LAR program is pretty straightforward. As described in Subsection 4.2.1, each definition of the LAR program corresponds to a C function accepting in fact only one argument: a lazy activation record which encapsulates the arguments of the defined function in the LAR program. In a C function the `ACTUAL` operator follows its argument's *prev* entry, which is a pointer to the argument of the caller function. Lazy activation records can be easily implemented as structures in C in accordance with the description in Section 4.1. Figure 4.2 presents part of the prelude code generated by GIC's C code generator which accurately reflects the ideas mentioned above. Some further explanations on this code follow:

- The `T_` structure, which implements LARs, has two extra fields (`arity`, and `nesting`) holding the LAR's arity (i.e. how many arguments are encapsulated in the LAR or, in other words, the corresponding function's arity in the FOL program) and the size of the `nested` list of contexts (i.e. how many nested `case` expressions there are in the body of the corresponding function in the FOL program). This makes it possible to determine the exact size of a particular LAR during execution—GIC's experimental garbage collector takes advantage of it in order to store the LARs in memory unwrapped. Observe that `arity` is also used by some field-selecting macros.
- The `AR_S` macro allocates a LAR on the stack. There is also its counterpart, the `AR` macro (not displayed here), which allocates a LAR on the heap. The expansion of the `LAR_STRUCT` macro is an anonymous structure used for type-castings.
- Recall from the description of NVIL (Subsection 2.2.2) that the *EVAL* function returns either a ground value or a pair containing a constructor and a context. The `Susp` structure implements exactly this kind of pairs. This is also the return type of the C functions generated for the definitions of a LAR program (regardless of whether the defined names are nullary variables or functions in the LAR program). Given the name `x` of a C function, the `FUNC(x)` or `VAR(x)` macro expands to this function's declaration (header).

In order to get a better idea of how the C generated by GIC looks like, we can extend Example 4.2.1 to see the C functions that correspond to the definitions in the LAR representation.

Example 4.2.2. Figure 4.3 shows the LAR program from Example 4.2.1 and the corresponding C functions, declarations, and macro definitions generated by GIC's LAR back-end. Note that, for the reader's convenience, only the C code that implements the intensional definitions is presented, the rest

```c
typedef unsigned char byte;

typedef struct T_* TP_;

typedef struct Susp {
  int constr;
  TP_ ctxt;
} Susp;

typedef Susp (*LarArg)(TP_);

typedef struct T_ {
  TP_ prev;              // link to parent LAR
  byte arity;            // the number of arguments in this LAR
  byte nesting;          // the number of nesting links
  void* data[];          // the rest of this struct contains:
                         //   - array of args to evaluate (ARGS)
                         //   - computed thunk values (VALS)
                         //   - nested contexts (NESTED)
} T_;

#define LAR_STRUCT(n_arity, n_nesting)            \
  struct {                                        \
    TP_ prev;                                     \
    byte arity, nested;                           \
    LarArg the_args[n_arity];                     \
    Susp the_vals[n_arity];                       \
    TP_ the_nested[n_nesting];                    \
  }

#define THE_ARGS(T)   ((byte *) &((T)->data))
#define THE_VALS(T)   (THE_ARGS(T) + (T)->arity * sizeof(LarArg))
#define THE_NESTED(T) (THE_VALS(T) + (T)->arity * sizeof(Susp))

#define ARGS(x, T)    (((LarArg*) THE_ARGS(T))[x])
#define VALS(x, T)    (((Susp*) THE_VALS(T))[x])
#define NESTED(x, T)  (((TP_*) THE_NESTED(T))[x])

#define VAR(x)        FUNC(x)
#define FUNC(x)       Susp x(TP_ T0)
#define ACTUAL        T0 = T0->prev

#define GETARG(x, T)  ({                      \
      if (ARGS(x, T) != NULL) {               \
        Susp val = ARGS(x, T)(T);             \
        VALS(x, T) = val;                     \
        ARGS(x, T) = NULL;                    \
      }                                       \
      VALS(x, T);                             \
    })

#define AR_S(n_arity, n_nesting, ...)             \
  ((TP_) &((LAR_STRUCT(n_arity, n_nesting))       \
    { T0, n_arity, n_nesting, { __VA_ARGS__ } }))
```

Figure 4.2: Implementing LARs in C

```
result    = f f_x__0 f_y__0 + f f_x__1 + f_y__1
f f_x f_y = g g_z__0 + f_y
g g_z     = g_z
f_x__0    = ACTUAL.4
f_x__1    = ACTUAL.5
f_y__0    = ACTUAL.6
f_y__1    = ACTUAL.9
g_z__0    = ACTUAL.f_x + 1
```

```
FUNC(result);
FUNC(f);
FUNC(g);
VAR(f_x__0);
VAR(f_x__1);
VAR(f_y__0);
VAR(f_y__1);
VAR(g_z__0);
#define f_x(T0) GETCBNARG(0, T0)
#define f_y(T0) GETCBNARG(1, T0)
#define g_z(T0) GETCBNARG(0, T0)

FUNC(result){
  return ((Susp) { (f(AR_S(2, 0, f_x__0, f_y__0)).constr+
                    f(AR_S(2, 0, f_x__1, f_y__1)).constr), NULL });
}
FUNC(f){
  return ((Susp) { (g(AR_S(1, 0, g_z__0)).constr+f_y(T0).constr), NULL });
}
FUNC(g){
  return g_z(T0);
}
VAR(f_x__0){
  ACTUAL;
  return ((Susp) { 4, NULL });
}
VAR(f_x__1){
  ACTUAL;
  return ((Susp) { 5, NULL });
}
VAR(f_y__0){
  ACTUAL;
  return ((Susp) { 6, NULL });
}
VAR(f_y__1){
  ACTUAL;
  return ((Susp) { 9, NULL });
}
VAR(g_z__0){
  ACTUAL;
  return ((Susp) { (f_x(T0).constr+((Susp) { 1, NULL }).constr), NULL });
}
```

Figure 4.3: C code for LAR definitions

of the code in the final C program (for example, prelude code, initialization code, main function, etc.) has been excluded.

Notice that *all* C functions return a value of type `Susp` as a result, even in the case that no user-defined data types are involved. However, we actually have this uniform behaviour at a low cost: if the function returns a ground value then it simply stores it in the place of the constructor and sets the context (the `ctxt` field) to `NULL` in its result.

The C code also illustrates a simple optimization performed by the LAR back-end. When the value of a formal argument is required only once in the function's body there is no need to store the computed value in the function's LAR. In this case the macro `GETCBNARG` is used instead of `GETARG`, which is defined as:

```
#define GETCBNARG(x, T)     (ARGS(x, T)(T))
```

No user-defined data types are used in this example, therefore, as expected, all LARs are allocated on the stack.

Chapter 5

C stack elimination

As described in Chapter 4, GIC's LAR back-end uses C functions as the basic unit for the output code; it produces one C function per LAR definition. This means that the resulting executables make extensive use of the C call stack, which arguably has many disadvantages. In an attempt to solve multiple issues at once, the first objective of this work was to abolish the C call stack and replace it with a custom stack.

A more ambitious alternative (which would attack even more issues at once) would be to build a new code generator that emits LLVM code. The main reasons for not following this approach was the amount of required work (taking into account the many more low-level details that would have to be settled in comparison with the existing C generator) and the uncertain (at least in the short-term) benefits for execution speed and garbage collection that such an approach would offer.

The rest of this chapter describes an attempt to eliminate the use of the C stack, presents the results, and tries to explain the main reasons for failing to meet the objective of faster execution. The uncertain balance between the pros and cons of this approach led to the rejection of the solution; however, the experience gained is potentially useful.

5.1 Motivation

The need for better support for garbage collection, along with the hope of gaining some execution speed, were the main motives for getting rid of the C stack. Additionally, such a modification would probably not require any deep cuts in GIC's LAR back-end. Sticking with C as the target language seemed reasonable in terms of feasibility, portability and execution speed of compiled programs. More specifically, some important benefits of the approach would possibly be the following:

- Better support for garbage collection: it is relatively easy to make out the root set in the case of a custom stack, where the independence from the possible optimizations of a C compiler can be guaranteed. This is probably helpful in any case, however it is crucial particularly in the case of accurate garbage collection.
- Avoid stack space limitations imposed by the operating system and, more generally, enhance portability.
- Possibly faster execution: using jumps instead of function calls and maintaining an appropriate custom stack seemed to provide a good chance for generating faster-running code.
- Direct implementation of optimizations: a custom stack (along with a custom calling convention) provides the opportunity of explicitly designing, implementing, and guaranteeing specific optimizations (such tail-call elimination, etc.) independently from the C compiler.

Moreover, in the long term, ideas proved to be successful in the setting of the C code generator could also be transferred to a new low-level code generator.

5.2 Implementation outline

In order to quickly test the idea of maintaining a custom stack, some small and simple Haskell programs were selected and compiled with GIC. Then, for each outputted C program, a version using a custom stack was coded by hand. This version was based on the code generated by the compiler and roughly admitted the following modifications:

- Each function in the original C program becomes a label.
- Each function call in the original C program becomes a jump (`goto`) to a statically known label and each function return becomes a jump to a statically unknown destination (implemented as an "assigned `goto`", which is a gcc-specific feature).
- `LarArg` becomes a code pointer (pointer to a label instead of pointer to a function)
- A stack entry consists of:
  - a `Susp` value, which is the function's return value;
  - a code pointer, which is the function's return address;
  - and a `TP_` value, which is the function's context.

In fact, other flavors of stack where also tested; however, representing the stack as a pointer to a structure conforming with the description above seems to be the most efficient implementation. As an example, in another scenario, the stack entry is a union which contains either a `Susp` value or a return address and a context. In this case the stack frame is overwritten before jumping to the return address, which is cached locally.

In all cases, stack entries played also the role of local storage: stack entries filled with return values are under the responsibility of the "caller" (i.e. the block of code that initially pushed them in the stack).

Example 5.2.1. In order to get a better picture of how the output C programs look like after the modifications described earlier, we can use the following simple FL code as an example:

```
result = fib 22
fib x = if x<2 then 1 else (fib (x-1)) + (fib (x-2))
```

The C functions produced by GIC that correspond to the definitions above are presented in Figure 5.1. Figure 5.2 presents the corresponding part of the modified program. The modified definitions of `LarArg` and `ACTUAL` (see Figure 4.2 in Subsection 4.2.3 for a comparison) and the definition of the type of the stack frames are:

```
typedef void * LarArg;

#define ACTUAL        cst_top->ctxt = cst_top->ctxt->prev

typedef struct RRC {
  void * ret_addr;
  Susp ret_val;
  TP_ ctxt;
} RRC;
```

```
FUNC(result){
  return fib(AR_S(1, 0, fib_x__0));
}

FUNC(fib){
  return (
    ((Susp) { (fib_x(T0).constr <
               ((Susp) { 2, NULL }).constr), NULL }).constr?
    (((Susp) { 1, NULL })):
    (((Susp) { (fib(AR_S(1, 0, fib_x__1)).constr +
                fib(AR_S(1, 0, fib_x__2)).constr), NULL }))
  );
}

VAR(fib_x__0){
  ACTUAL;
  return ((Susp) { 28, NULL });
}

VAR(fib_x__1){
  ACTUAL;
  return ((Susp) { (fib_x(T0).constr-((Susp) { 1, NULL }).constr), NULL });
}

VAR(fib_x__2){
  ACTUAL;
  return ((Susp) { (fib_x(T0).constr-((Susp) { 2, NULL }).constr), NULL });
}
```

Figure 5.1: C functions produced by GIC for the FL code in Example 5.2.1

The stack-associated functions can be implemented as follows:

```
inline void push_rrc(void * ret_addr, TP_ ctxt) {
  cst_top--;
#ifdef SAFE
  if ((byte*)cst_top + MAXSTACKSIZE < (byte*)cst_bot) {
    printf("stack overflow!\n");
    exit(1);
  }
#endif
  cst_top->ret_addr = ret_addr;
  cst_top->ctxt = ctxt;
}

inline st_entry * pop_rrc() {
  st_entry * tmp = cst_top;
  cst_top++;
#ifdef SAFE
  if (cst_top > cst_bot) {
    printf("stack already exhausted!\n");
    exit(1);
  }
#endif
  return tmp;
}
```

```
rslt: {
  Susp result;
  push_rrc(&&l1,(AR_S(cst_top->ctxt, 1, 0, &&fib_x__0)));
  goto fib;
l1:
  result = pop_rrc()->ret_val;
  cst_top->ret_val = result;
  goto *(cst_top->ret_addr);
}

fib: {
    Susp result;
    if (((Susp) { (fib_x(cst_top->ctxt,l5).constr <
                  ((Susp) { 2, NULL }).constr), NULL }).constr) {
      cst_top->ret_val = ((Susp){ 1, NULL });
    } else {
      push_rrc(&&l2,(AR_S(cst_top->ctxt, 1, 0, &&fib_x__1)));
      goto fib;
l2:
      CLEAR_CTXT(1);
      push_rrc(&&l3,(AR_S((cst_top+1)->ctxt, 1, 0, &&fib_x__2)));
      goto fib;
l3:
      CLEAR_CTXT(2);
      result = (Susp){ ((pop_rrc())->ret_val).constr +
                       ((pop_rrc())->ret_val).constr, NULL };
      cst_top->ret_val = result;
    }
  goto *(cst_top->ret_addr);
}

fib_x__0: {
  ACTUAL;
  cst_top->ret_val = ((Susp){ 28, NULL });
  goto *(cst_top->ret_addr);
}

fib_x__1: {
  ACTUAL;
  cst_top->ret_val = ((Susp) { (fib_x(cst_top->ctxt,l6).constr -
                                ((Susp) { 1, NULL }).constr), NULL });
  goto *(cst_top->ret_addr);
}

fib_x__2: {
  ACTUAL;
  cst_top->ret_val = ((Susp) { (fib_x(cst_top->ctxt,l7).constr -
                                ((Susp) { 2, NULL }).constr), NULL });
  goto *(cst_top->ret_addr);
}
```

Figure 5.2: Elimination of the C stack

Note that there are many possible ways of implementing the custom stack and the associated functions; the set of definitions presented above is just one out of many that were tested for the evaluation of this approach.

5.3 Results

As mentioned earlier, many different versions of stackless programs were derived and tested. The differences had to do with the implementation of the stack structure and the associated *push* and *pop* functions as well as with the amount of compiler directives (structure alignment, variable caching in registers, etc.) used in the code. Moreover, both `gcc` and `llvm-gcc` were used on x86 and x86-64 machines for the testing with a few different combinations of optimization flags.

Surprisingly perhaps, the original version of a program ultimately proved to be always faster than all its stackless counterparts! In the case of our example from the previous section (Example 5.2.1) the fastest stackless version runs almost 47% slower than the original program produced by GIC when we compile with `gcc` version 4.4.5 (passed the `-O3` flag) and run on a machine with four quad-core Intel Xeon E7340 2.40GHz processors and 16 GB memory, running Debian 6.0.7.

Although this might seem strange at first, there are some possible explanations for this behaviour, which are presented in the next section.

5.4 Conclusion

The slower execution times and the difficulty to produce even by hand competitive stackless programs were understood as hints that the design (characterized by the use of assigned `gotos`) might not be the best one. Therefore the solution of eliminating the stack this way was ultimately rejected.

5.4.1 Interpreting the results

Some observations that might support, up to some degree, the results presented in the previous section are the following:

- GCC, following the C calling convention for the x86-64 architecture and also optimizing the calls in any case, passes some arguments in registers when a function is called. It seems to be impossible to implement something similar in the case of our custom stack using only plain C (and remaining portable). In the case of the stackless programs, all arguments and return values are always written on and read from the custom stack. This seems to be a non-negligible source of execution slowdown.

- Assigned gotos used in the custom implementation are translated, in the case of x86, to unconditional `jmp` instructions with a register operand (or memory operand for GCC versions 4.1 and 4.3). This could be slower than using the `call` instruction (which jumps to a known destination) or than using the specialized `ret` instruction.

- The custom implementation possibly imposes a negative effect on register allocation. The role of `ESP` and `EBP` (or `RSP` and `RBP` in the x86-64 case) is degraded, effectively leaving the allocator with fewer registers to do an equivalent job—as the custom stack also requires maintenance.

Certainly, there can be many other reasons that might go as deep as the details of the underlying hardware architecture —after all, the testing was performed on the x86 and x86-64 architectures which are closely related. Moreover, the testing was by no means thorough in terms of the use of the optimization flags of GCC.

5.4.2 Useful experience

Some potentially useful experience derived from this small experiment includes the following:

- The feasibility of eliminating the C stack usage for the output programs of GIC was confirmed. Moreover, it is very likely that it would require only small changes in the compiler's C back-end.
- All the reasons for the decreased execution speed that were presented earlier seem to have to do with low-level details over which we have almost no control at the C level. This questions the idea of getting rid of the C stack in the case that C remains the target language of the back-end. Other ways of overcoming some of the difficulties imposed by the C stack should also be investigated.
- A solution that appears even more appealing now is that of a new code generator. It is evident that what is needed is more low-level than C and, considering also the importance of portability which is a high priority of GIC, LLVM seems to be a reasonable choice. Note that some of the ideas used in the experiment of eliminating the C stack usage could also find their way into a new code generator, where they will be possibly supported by the provided control over low-level features.

Chapter 6

Boehm-Demers-Weiser Garbage Collection

This chapter provides a very brief review on garbage collection with the focus being on the case of functional programming languages, discusses in particular the needs of GIC in garbage collection, and argues that the Boehm-Demers-Weiser garbage collector is a viable solution. Finally, it describes the integration of this collector in the project. For the rest of the chapter, we assume the case of uniprocessor systems.

6.1 Introduction

Garbage collection refers to any method which automatically (i.e. without the programmer's intervention) reclaims memory in a safe manner [Wils92]. Such a method was first described by McCarthy [McCa60] and, since then, several forms and variations of garbage collection have been invented and implemented.

The basic idea behind garbage collection is that an approximation of the set of the heap-allocated data that is still needed by a running program at some point of the execution is the set of *reachable* data. That is, we can approximate the *live* data (the exact calculation of which is undecidable) with the data that can be reached by starting from a root set (which includes the statically allocated space, the execution stack, and the machine registers—and which is certainly reachable for the program) and following an arbitrary number of pointers to heap-allocated data.

In general, garbage collection aims at both safe and efficient automatic memory management. The need for garbage collection may range from optional or auxiliary to (almost) absolutely necessary, according to the particular setting (i.e. the programming language and the facilities it provides, the presence of modules, the structure or organization of a specific project, etc.)

The major benefits that may be expected from the use of garbage collection are summarized below:

- Garbage collection substantially supports fully modular programming by eliminating unnecessary intermodular dependencies [Wils92].
- It relieves the programmer from the hard and error-prone task of manual memory management—this is crucial particularly in the case of high-level languages. For example, note that explicit memory deallocation is not easy to be integrated in the functional programming style, let alone that it could discourage optimization such as tail-call elimination. Moreover, notice that in a functional language memory can also be implicitly allocated (for example, to store closures) which poses additional and probably more important problems.
- Certain kinds of bugs and memory leaks become impossible under garbage collection. Memory leaks in particular can be very hard to detect whereas their absence is highly important for long-running programs (such as simulation programs, scientific computation programs, server applications, etc).

- In practice, a program running with garbage collection is usually competitive with its counterpart running with explicit memory deallocation [Wils92]. Under some circumstances, garbage collection can be even cheaper than explicit deallocation [Appe87].

In practice, most high-level language implementations come with an intergraded garbage collector, specifically designed for being a part of a particular runtime environment. In such a case, the garbage collector usually cooperates with other parts of the runtime environment. For example, a strong static type system can provide information on the size of objects and the exact location of pointers or references to other objects, effectively eliminating the need for tags or descriptors [Appe89, Gold91]. Moreover, knowledge on the system's expected behaviour (i.e. the expected frequency of memory allocation, the expected lifespan of most allocated objects, the presence or absence of mutable fields, the expected size of objects, etc.) may also be used to fine-tune such a specialized garbage collector.

Nevertheless, implementing an "agnostic" garbage collector, which assumes nothing about the runtime environment, may have its own benefits. Some of these benefits are summarized below:

- Such a collector can be used in almost any setting. It can be used with any existing compiler and programming language (even with those completely unaware of garbage collection such as C) and can provide existing code with garbage collection even if this code was originally designed to use explicit memory deallocation. As it does not require any kind of tags or headers, it allows for using standard machine representations of data and, therefore, makes interfacing with existing libraries and the underlying operating system easier [Boeh88].
- The absence of tags also leads to smaller memory footprints and faster operations on values of ground types. Generally, only a small runtime overhead is imposed to programs that do not make intensive use of the heap—i.e. when only a small amount of heap memory is used by a program then a version of this program using the garbage collector should not be substantially slower than a version that does not.
- The design of a compiler, which is to be combined with this kind of collector, is simplified. On the contrary, a specialized garbage collector typically depends on the preservation of specific invariants from the part of the generated code. Generating code that conforms with the assumptions made by the garbage collector can be a difficult and error-prone task, which is avoided altogether in the case of an agnostic collector [Boeh88].

Assuming nothing about the runtime environment basically implies that the garbage collector does not have any information about which memory locations contain memory addresses, i.e. bit patterns intended to be used as such. This naturally leads to some *conservative pointer-finding* scheme: any properly aligned bit pattern that could be an address of a heap-allocated object is considered to be truly such an address by the garbage collector. More details on such collectors (often referred to as *conservative garbage collectors*) will be given in the rest of this chapter.

6.2 Basic categories of garbage collectors

There are many ways to form some basic categories of garbage collectors. The possible criteria range from the underlying algorithm to distinguishing technical features—usually there are no sharp lines between them. For our purposes, we will use a simple, intuitive, and somewhat custom categorization, which is visualized in Figure 6.1 (arrows indicate some kind of specialization, dotted arrows indicate further optional specialization—a full path consisting of opaque arrows gives a complete garbage collection specification). Note that we only use distinguishing features related with the operation or the implementation of garbage collectors for this categorization, i.e. we do not take into account the way

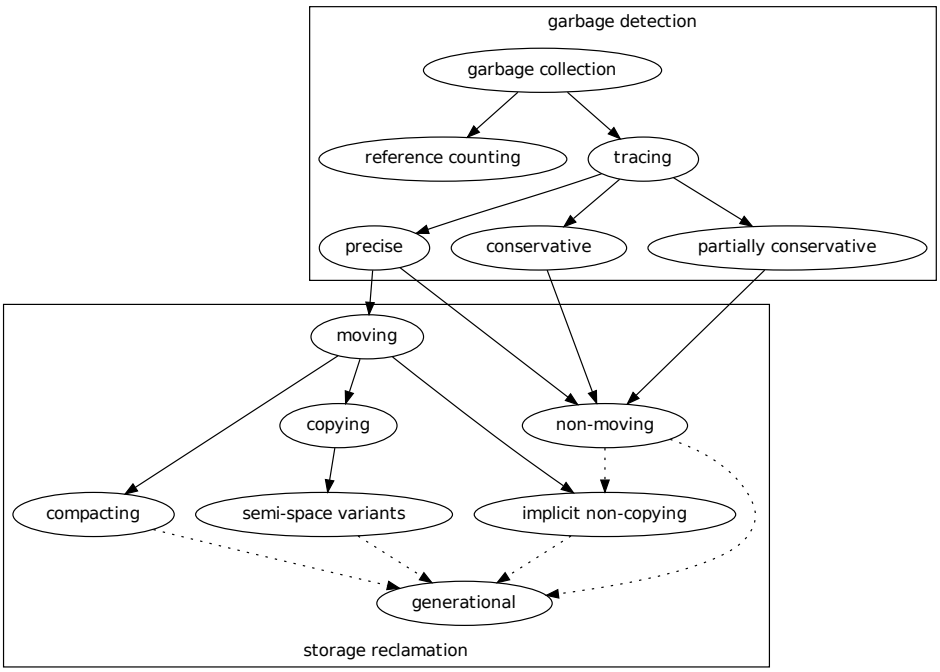

Figure 6.1: Basic categories of garbage collectors.

garbage collection algorithms are related to each other theoretically. For example, as we shall see in Subsection 6.2.2, non-moving implicit collection in fact generalizes the idea of copying collection, and the operations of the corresponding implementations are (abstractly) isomorphic to each other. However, this important relationship is not illustrated in this simplified diagram. Instead, from the (concrete) operational point of view, non-copying implicit collectors can be considered as a subcategory of non-moving collectors, i.e. collectors that do not move live data in memory, because they can be implemented as such. Moreover, some important categories of collectors (such as incremental collectors) as well as many hybrid designs do not appear in the diagram, as they fall out of the scope of this work.

We assume that in general a garbage collector has two basic abstract functionalities: (i) garbage detection, and (ii) storage reclamation—note, however, that while some algorithms for garbage collection realize these functionalities as two separate phases others integrate them together. In Subsections 6.2.1 and 6.2.2 we review the available options for realizing these functionalities, and we do this in a manner very close to what we would do if we had to chose the most appropriate kind of collector for a given application (having mostly GIC in mind).

6.2.1 Detecting garbage

Tracing garbage collectors and reference counting systems

Tracing collectors are characterized by the way they distinguish between live data and garbage: when there is need for reclaiming storage, they find live data by literally starting with the root set and traversing through allocated data by following pointers. The mark-sweep algorithm [McCa60] (which was probably the first algorithm for garbage collection to be invented) and its variants as well as copying collectors fall into this category. The differences between them have essentially to do with

how exactly they reclaim storage. On the other hand, reference counting techniques [Coll60] detect garbage in a different way: every allocated object is associated with a counter that stores the number of references to the object. The counter is incremented when a reference to the object is created, and is decremented when a reference to the object is removed. If at some point during execution the counter becomes zero (indicating that there are no references to the object under consideration any more and therefore the object is unreachable) then the memory allocated for this object can be reclaimed. Reference counting systems undoubtedly enjoy some strong points such as, for example, the inherently incremental nature of their operation which makes them suitable even for real time applications. However they also pose weaknesses making them unsuitable for systems such as GIC: (i) they cannot reclaim circular structures and usually rely on other garbage collection techniques for reclaiming such structures from time to time, and (ii) the total runtime cost of reference counting is usually high because reference counting techniques continuously track *all* mutations of addresses stored in data (in contrast with tracing techniques which essentially work with "snapshots"). We will not consider reference counting techniques in this chapter any further.

Precise and conservative pointer-finding garbage collectors

After deciding to use a tracing collector (i.e. after deciding on the general method to use for detecting garbage), the next thing to do could be to settle the details for the functionality of garbage detection by deciding how the collector should find the root set and the pointers stored in data structures. As we mentioned earlier, it is possible for a compiler to communicate such information to the garbage collector—in the simplest case this can be done by using bits to tag the data so that the collector can distinguish between pointers and non-pointers, but there are also several other ways. Garbage collectors that use such information in order to accurately distinguish pointers are often called *precise*. We have also described earlier an alternative, namely conservative pointer-finding collectors, where the garbage collector uses no external information about pointers and treats every (aligned) bit pattern as a potentially valid pointer. This approach has benefits (see Section 6.1) which follow exactly from the fact that no assumptions are made for the rest of the runtime system, and which could be summarized in the statement that conservative pointer-finding collectors provide an easy way to get efficient garbage collection in the average case. A serious inherent drawback is that the safety of such systems can be compromised by language facilities such as casting pointers to integers and vice-verse [Wils92] or unchecked pointer arithmetic: such operations can be performed either by programmers or by some optimizing compilers and they effectively "hide" pointers from the garbage collector which still can be restored at a later point through arithmetic or binary operations. However, it has been shown that simply conforming with a set of constraints assures that this problem cannot arise [Boeh92, Boeh96].

As conservative garbage collection turns out to be a reasonable choice in our particular case (for reasons explained in Subsection 6.3.1), and its major strong points have already been described, we will now focus briefly on the downturn of this approach, which has to do with the fact that conservative pointer-finding collectors actually make *two* approximations: First, they approximate the set of live data at some point during execution with the set of reachable data at this point, which is what every collector does. Second, they compute a superset of the set of reachable data: besides addresses that were intended to be used as such, also bit patterns that can be interpreted as valid heap memory addresses contribute to the data that are considered reachable by these collectors.

A first significant implication of the latter approximation has to do with the potential failure of conservative pointer-finding collectors to reclaim memory due to misidentified pointers. For example, an integer value could be interpreted by the collector as the address of a heap-allocated object which also happens to be unreachable at this point of execution and therefore should normally be reclaimed. The misidentified pointer, however, prevents the object's deallocation and leads to memory leaks that theoretically could result in excess memory usage [Hast91, Boeh02]. A strong manifestation of this problem is considered to be unlikely in practice, although some negative results have shown up

[Edel92, Went90]. Note that, in any case, the probability of such misidentification increases with the size of the heap and decreases with the width of memory addresses (in many architectures this is the same as the size of the processor's word). There are relatively simple techniques that substantially reduce the probability of misidentifications in practice [Boeh93]. A theoretical result has also been derived, which assures that, under reasonable assumptions regarding the operation and usage of a conservative pointer-finding collector, if all data structures that are used by a program adhere to specific fairly common properties then a bounded number of misidentified pointers can only result in a bounded amount of leaked memory [Boeh02]. Note that merely treating the heap accurately (for example, using object descriptors) suffices for getting a theoretical bound for the possible erroneously retained memory due to misidentified pointers, as the call stack is always bounded in practice.

A second implication of conservative pointer-finding is that, in general, allocated objects cannot be moved in memory—at least those objects that are pointed from conservatively scanned areas of memory, usually the stack: moving objects in memory requires the update of the pointers pointing at them, but updating a misidentified pointer definitely breaks program correctness, which, in turn, may yield unpredictable results. We shall discuss this issue further in the following subsection.

In the above paragraphs we have already referred to hybrid garbage collection systems [Bart88, Bart89a, Detl90, Sche88], though not explicitly, which usually rely on conservative pointer finding for scanning the stack while they treat the heap precisely. This approach (i) reduces conservatism and, along with it, the probability of serious memory leakages, and (ii) makes available for conservative-based collectors effective techniques that were normally designed for precise garbage collection. Some of these techniques are mentioned in 6.2.2 and usually some modifications are required to adapt them in a partially conservative system. It is evident that hybrid systems fit well in partly cooperative language implementations such as GIC, where enough information for accurately treating the heap can be easily extracted and communicated to the garbage collector [Bart89b, Sche88].

6.2.2 Reclaiming storage

After marking reachable data all remaining data is garbage and can be safely deallocated. The storage reclamation functionality is more low-level in comparison with the marking functionality. Some decisions related with the marking phase (such as conservatism) can have an impact on the storage reclamation functionality. However, as shown in Figure 6.1, in the case of precise garbage collectors, storage reclamation is largely independent from anything else and can be designed from scratch, freely implementing (or mixing up) the available techniques. This is where bothering to have the compiler cooperate with the garbage collector pays back.

Moving and non-moving garbage collectors

Non-moving garbage collectors are those which do not move reachable objects in memory. As mentioned earlier, (fully) conservative pointer-finding collectors fall into this category. In the case of non-moving collectors storage reclamation can be implemented easily (usually a free list is used which links all reclaimed objects), and the original spatial locality of objects is preserved—however, temporal locality can still be a problem as newer objects are gradually placed among older objects [Wils92]. A major problem of this approach is memory fragmentation, which results from the different sizes of objects. For non-moving collectors the extend of memory fragmentation depends entirely on the effectiveness of the accompanying allocator and the memory request stream of the running program. Good placement policies, separate free lists for objects of different size, merging of adjacent reclaimed spaces, and heuristics estimating the objects' possible lifespan are some methods that can partially compensate for the problem of fragmentation at the cost of some additional complexity for the im-

plementation [Wils95]. In practice, the fragmentation observed for most programs running under a non-moving garbage collector can be considered acceptable.

A second potential problem with non-moving collectors is the fact that garbage collection cost is proportional to the total size of the heap. For example, (the original) mark-sweep touches all reachable data during the mark-phase and all garbage during the sweep-phase (possibly also re-touching reachable data). However, there are available techniques for combating this problem. One effective technique (non-copying implicit collection) is derived by generalizing the idea behind copying garbage collection and is discussed in the next paragraphs.

Moving garbage collectors move reachable objects in order to efficiently combat the important problem of fragmentation, which is inherent to non-moving collectors. Compacting collectors move objects so that that they take up a continuous memory region, usually by sliding them so that each one of them becomes adjacent to the previous one. Additional benefits of this approach include the possibility of fast allocation, as new objects are consecutively allocated in a continuous free memory space, and possibly improved locality, as old and new objects do not mix up spatially. Copying collectors eliminate fragmentation but also bear the advantageous property that they do not collect garbage explicitly. The most common realization is the “semi-space collector” [Feni69], which splits available memory in two halves (often referred to as “from-space” and “to-space”) and copies reachable objects from the “from-space” to the “to-space” as these are reached by the traversal (which can also be preformed iteratively [Chen70] instead of recursively). After this operation is concluded the “to-space” contains only the reachable data placed compactly next to each other, and the roles of the two spaces is flipped for the next collection cycle. For copying collectors, the garbage collection cost is proportional to the size of reachable data: increasing the total amount of available heap memory makes a copying collector arbitrarily efficient [Appe87], as the ratio of the size of reachable data to the size of total memory asymptotically approaches zero. Copying collectors, like compacting collectors, make allocation a very fast operation, which usually amounts in just advancing a pointer pointing at the beginning of the free memory area of the currently active semi-space. However, the locality of data may be damaged as these collectors rearrange the configuration of objects. Copying large reachable objects and the fact that only one semi-space is active at any given time during the client program execution can also be considered as deficiencies of copying collectors.

Non-copying implicit garbage collection [Bake92] is based on an idea very closely related to copying garbage collection—although this may not be apparent if one just compares implementations of these two kinds of collectors. The key observation is that the spaces of a copying collector are just one possible realization of sets of data i.e. the “from-set” and the “to-set”. If we implemented these sets as doubly-linked lists then, instead of copying objects from from-space to to-space, we would have to unlink objects from the from-set and link them in the to-set. Any other (efficient) implementation of sets would also do. Note that, in any case, it is still possible to quickly find out which set each object belongs to by using a flag for every object. As in the case of copying collectors, space reclamation is also implicit for non-copying collectors and the garbage collection cost is proportional to the number of reachable objects. Although the worst case asymptotic time complexity is the same as for copying collectors, in the presence of sufficiently large reachable objects (especially if these do not contain pointers) non-copying implicit collectors may have an advantage in practice. Moreover, as non-copying implicit collectors fall into the more general category of non-moving collectors, they could be used in combination with conservative pointer-finding—precise treatment of the heap would also retain the advantage that large objects without pointers do not even need to be scanned. On the downside, these collectors (i) inherit the problem of fragmentation, which is present in all cases where data cannot be moved in memory, (ii) they possibly impose a small time overhead for the allocation operation in comparison with the copying collectors, and (iii) they also impose a small space overhead for wrapping the objects appropriately for the chosen implementation of sets.

Generational garbage collectors

Generational garbage collectors [Lieb83, Unga84] aim at improving garbage collection efficiency and have also been used in real-time settings as an alternative to incremental collectors. The key point of their design is that objects of different ages are treated differently by these collectors. The following two observed trends in the runtime behaviour of programs indicate what the age of an object should mean for the garbage collector:

1. Objects tend to "die" young, i.e. it is more likely for a recently allocated object to soon become unreachable (and thus reclaimable) than it is for an object that has already been through at least one garbage collection cycle.

2. Usually newer objects contain references to older objects; the opposite should be rare. Indeed, the only way to create a reference from an older object to a newer one is an in-place (catastrophic) update operation to a field of the older object, which is not very usual for non-imperative languages.

These observations suggest that the garbage collector should spend more effort on reclaiming storage occupied by recently created (and, therefore, presumably short-lived) objects, and save time by only infrequently consider older objects (which have already survived a small number of garbage collection cycles). Usually, such functionality is realized by partitioning allocated objects in two or more sets (generations) and start by garbage collecting the set representing the first generation. Objects that survive a number of collections are promoted to the second generation set, which fills up slowly only due to promotions from the first generation and therefore is much more infrequently garbage collected. This process is repeated in scale if the collector supports more generations.

At first generational techniques were usually combined with copying collectors: indeed, the strain on a copying collector can be relieved when older, long-lived data can be left aside for long time periods instead of being continuously copied. Efficient, compact-sized systems of this kind have been described and successfully implemented [Appe88] to work in demanding environments. Nevertheless, generational techniques can also be (and actually have been) combined with other kinds of collectors; notably, they can be integrated even in conservative pointer-finding collectors [Bart89a, Deme90].

It is necessary for generational collectors to have a way of detecting (presumably rare) intergenerational pointers from older to newer generations, as these must be taken into account during the garbage collection of the newer generation. Many solutions have been proposed for this issue—Whilson includes a detailed presentation in [Wils92]. However, no matter how efficient these solutions might be, it is evident that if, for any reason, some program fails to conform with the behaviour that is expected in accordance with the second assumption generational collectors adopt then, under some circumstances, the benefits of generational collection can be negated by the penalties due to frequently falling in a case which should normally be rare (and therefore is more expensive than what is considered the common case). More generally, generational collectors, as they heavily depend on heuristics, are amenable to runtime behaviours that diverse from what is considered typical. Nevertheless, they tend to be efficient indeed in practice.

6.3 Garbage collection in GIC

In this section we shall review the current needs of GIC's LAR back-end in garbage collection and investigate the suitability of the garbage collector described in Section 6.3.2.

6.3.1 Options and restrictions

As we saw in Chapter 4, the LAR back-end currently generates portable C code which relies on the C call stack. Insisting on the use of the C stack currently leads to better execution time of compiled programs (Chapter 5) but essentially imposes a serious restriction for the garbage collector: the C stack must be scanned for roots conservatively [1]. In fact, compiling programming languages that require garbage collection into C can be considered as a motivation for developing conservative pointer-finding collectors [Boeh93].

As shown in Figure 6.1 and explained in Section 6.2, conservative pointer-finding implies in general that allocated objects cannot be moved in memory, thus excluding copying and compacting collectors. We still have a choice on whether the heap is also treated conservatively or information is kept on the location of pointers in heap-allocated objects. The reader is reminded that the only possible heap-allocated objects are LARs; information on the exact size of each LAR and on the location of the pointers stored in it can be easily extracted during compilation. Incremental and/or generational collection is also possible—actually generational collection in particular could be an attractive choice since, when it comes to user-defined data types, the programs compiled with GIC follow the usual pattern of functional programs in terms of memory usage, intensively allocating small-sized short-lived objects.

6.3.2 The Boehm-Demers-Weiser garbage collector

The Boehm-Demers-Weiser garbage collector [Boeh88, Boeh] (Boehm GC for short) is an implementation of conservative pointer-finding garbage collection. It uses a variant of the mark-sweep algorithm, implements most of the optimizing techniques described in [Boeh93], and also includes various additional low-level optimizations. It provides incremental and generational collection [Deme90, Boeh91] and supports object-type descriptors for accurately treating the heap. Both of these latter features potentially concern functional programming language implementations and therefore are interesting also in our case. The collector's allocator segregates objects according to many criteria (small objects in particular are treated efficiently; this is also important for a functional language runtime) and takes advantage of virtual memory facilities provided by the underlying platform.

In general, Boehm GC is a mature garbage collection system which has undergone extensive tuning and has already been used in many projects (as a garbage collector or a leak detector) and language runtimes [Boeh]. Moreover, it provides a rich interface and exposes many configuration parameters, thus making experimentation easier.

6.3.3 Integrating the Boehm-Demers-Weiser garbage collector

From the observations of Subsection 6.3.1 and the description in Subsection 6.3.2 it becomes evident that Boehm GC is presently a good match for GIC. Under the current state of affairs, i.e. given the LAR back-end as it is, some strong points of choosing this particular collector are the following:

- Boehm GC supports all the theoretically available options mentioned in Subsection 6.3.1. That is, it supports object descriptors for accurately treating the heap and provides incremental and/or generational functionality.

[1] Note that technically it is possible to combine the use of C (and the C call stack) as a back-end's target with accurate pointer-finding [Hend02]. Henderson describes a technique that can be used to produce arbitrary C code where the roots can be accurately tracked. However, such an approach is certainly more complicated and difficult to implement than conservative pointer-finding collection; moreover, in terms of runtime performance, a comparison between this approach and the Boehm GC in particular seems to be indecisive [Hend02].

- This collector is a mature and up-to-date project, includes techniques derived from the latest developments in the field of conservative pointer-finding garbage collection, and supports extensive configuration. It has already been used in numerous projects and is widely considered to be a reliable and efficient solution.
- Its integration in GIC is fairly easy: only minor changes were needed, and these took place only in the C code generator. The generated C programs show only minor changes too. During the short development time there were no complications at all and GIC quickly obtained reliable garbage collection which, in turn, allowed for testing the compiler with more serious benchmarks.
- The portability of the generated programs is retained. Boehm GC uses specialized assembly code for performing some low level operations (such as root-finding) in many cases of well-known architectures, and also provides fallback operations that try to cope with unknown architectures.

Note that regardless of how the LAR back-end might evolve, the flexibility and reliability of Boehm GC make this collector a good choice in any case: it can be used as a point of reference for testing other collectors to be used with GIC or as a fallback garbage collector which quickly follows the evolution of the LAR back-end.

Fully conservative operation

For fully conservative garbage collection we only need to add a call to the collector's initialization routine in the `main` function of the generated programs and use the routines provided by the collector's interface for memory allocation. This simple solution works flawlessly for our collection of benchmarks. Currently, using Boehm GC for fully conservative garbage collection is the default option for GIC.

Although no abnormal memory usage has been observed so far, the suitability of fully conservative garbage collection needs further investigation in our case: Wentworth observes that any dynamically expanding data structure could cause serious memory leakage under conservative collection [Went90]. Lazy data structures are usually implemented this way, i.e. as structures that expand during execution in order to cache the computed values of the constructor's arguments, and this is also the case for lazy activation records. Lazy data structures are among those that do not conform with Boehm's assumptions for bounding memory leakage under conservative collection [Boeh02]. Note, however, that laziness does not actually cause the problem but rather magnifies it: a list data structure, implemented as a simply linked list in memory, with a misidentified pointer pointing at its head will cause a memory leakage no matter what, as the whole list will be retained indefinitely in memory. In a non-strict setting though it is possible that in fact there is no need at any time to keep the whole list in memory. As already mentioned, Boehm GC implements several techniques that reduce pointer misidentifications, which can be valuable in our case for avoiding the problems described above.

Partially conservative operation

It is also possible to use Boehm GC as a partially conservative collector. We certainly know the exact layout of an object (which can only be a LAR) at the time of its creation: this information is already present and used for allocating a LAR on the heap and can also be used for generating the corresponding object descriptor—actually we only need one descriptor per kind of LAR.

A comparison between the runtime behaviour exhibited by programs under partially conservative and fully conservative garbage collection shows that in our case Boehm GC is faster when in fully conservative mode. This is the primal reason for currently using fully conservative collection as the default

option in GIC. However, as mentioned in 6.3.3, fully conservative collection bares the hazard of serious memory leakage, and partially conservative collection significantly reduces the probability of such problems to arise. It remains to be checked in practice, possibly with a special suite of benchmarks, whether the benefits of partially conservative collection outweigh the faster execution times in the case of fully conservative garbage collection.

Generational operation

Boehm GC also provides an incremental/generational operation which can alternatively be used by GIC. Although some benchmarks seem to benefit in terms of execution speed from generational collection (speeding up 10-15%), most programs run slower (usually from 10% up to 30%) under this mode. The results are roughly the same regardless of whether the combined incremental-generational or the simple generational mode is used. In the former case, tweaking the number of partial collections between full collections does not seem to make any remarkable difference.

This is somewhat expected behaviour. On the one hand, functional programs typically do follow the pattern of creating many short-lived (and usually also small-sized) objects. On the other hand, we can immediately observe that laziness, in general, contributes to more frequent violations of both basic assumptions that make generational collection applicable (described in 6.2.2): computing something on demand involves updating some kind of reference (both for graph-reduction based implementations and for lazy activation records) and lazy data constructors make it a common case for older objects to point to newer objects (which were created later simply because they were created only at the time they were actually needed). Moreover, particularly in the case of lazy activation records, these newer objects cannot become unreachable before the older objects pointing at them do because all pointer fields of a LAR are assigned at most once. Therefore, the actual creation time of an object is not necessarily indicative of the object's "age" (with the meaning assigned to this term in the context of generational garbage collection).

6.3.4 Observations and suggestions

As we saw in Subsection 6.3.1, the use of C as the target language and the decision to keep using the C call stack significantly restricted the available options having to do with garbage collection because they call for conservative pointer-finding methods. At this point we may reconsider the role of the C call stack in garbage collection: eliminating the C stack and maintaining a custom stack would allow us in general to accurately scan the stack for roots. Compacting techniques would become easily applicable in this case; however, given the decrease in execution speed coming from the C stack elimination described in Section 5.3, the outcome in terms of performance is uncertain.

Taking a step further, we can see that in terms of efficient garbage collection a code generator targeting native code would probably be the best bet: full control over the calling convention and the (types of the) contents of registers would make all garbage collection techniques available. From the brief description in Section 6.2 it follows that especially copying garbage collectors would be an appealing choice for GIC in this case. Towards this direction, and although portability lies among the goals of this project, a native code generator for some popular architecture along with a custom (possibly copying) garbage collector could be implemented nonetheless, at least as a proof of concept. Useful comparisons could subsequently take place between the native code generator combined with the custom collector and (i) the existing C code generator combined with Boehm GC, and (ii) itself combined with Boehm GC. Even in the case of encouraging results, however, it is questionable whether this experience could also be used in the development of an LLVM back-end, which also retains portability, or native code generators are the only way to achieve high-performance garbage collection.

The possible benefits of using generational garbage collection in our case in particular is another issue. As described in Subsection 6.3.3, using the incremental/generational mode of Boehm GC seems to have on average a negative effect on the execution time of compiled programs. A possible explanation is that non-strictness (and our implementation using lazy activation records in particular) makes it quite common for older objects to point to newer objects. On the other hand, the rapid allocation of small short-lived objects along with the fact that all pointers to heap-allocated data in LARs are assigned only once (avoiding many costly updates of intergenerational pointers from older to newer objects) should normally improve the performance under generational garbage collection in our case. Both the results presented in 6.3.3 and these observations seem to suggest that the question of whether GIC could benefit from generational collection remains open. It seems possible that performance could be sensitive to the technical details of the specific implementation of generational garbage collection to be used with GIC. For example, efficiently handling a large number of intergenerational pointers from older to newer objects would be more useful than a faster update mechanism for such pointers which significantly degrades however as the number of pointers increases. Finally, the notion of "age" could be refined in order compensate for the "disturbance" introduced by the low-level details of the runtime system of lazy activation records: for example, it seems reasonable for LARs constituting a data structure to have the same age.

6.4 Conclusion

Garbage collection is not the focus of the this work. However, quickly providing a reliable solution to this issue was important because (i) it makes it possible to test GIC with more serious benchmarks (for which garbage collection was necessary) and compare it in terms of running time and memory consumption with popular Haskell compilers, (ii) it allows for testing the new features that have been scheduled for addition in GIC, (iii) it will be useful in the evaluation of any future custom garbage collector, and (iv) it was also a good opportunity both for an examination of GIC's runtime environment (and the possible alternatives) from a garbage-collection perspective, and for a first study of the related literature, which are presumably beneficial for making better decisions on the design or choice of any future collector.

Regardless of these benefits, Boehm GC should probably be considered as a temporary solution for GIC (at least as the default setting). As mentioned earlier, possible memory leakage in particular certainly requires further investigation as GIC's runtime falls in the category of "bad" contexts for conservative pointer-finding collectors. More important perhaps is the fact that (possibly in combination with a new low-level code generator) garbage collection could take advantage of the regularity of allocated objects (and call stack frames) imposed by the technique of lazy activation records. This suggests that developing a custom garbage collector for GIC is probably worthwhile.

Chapter 7

CAF memoization & LAR thinning

This chapter describes the implementation of one missing feature (CAF memoization) and one low-level optimization (LAR thinning) at the C level. The changes primarily target at better execution speed of compiled programs.

7.1 LAR thinning

This section describes a low-level optimization in the C implementation of LARs and in the machinery that provides access to the fields of this C structure. The changes resulted in better execution times and lower memory consumption for the compiled programs.

7.1.1 Motivation

Figure 4.2 (Subsection 4.2.3) shows the original C implementation of LARs used by GIC's C code generator. In comparison with the high-level description of LARs in Section 4.1, it is evident that the GIC's C back-end originally used two extra fields in the `TP_` (i.e. LAR) structure; namely the `arity` and the `nesting` fields. These fields make it possible to dynamically determine the size of an arbitrary LAR; this feature was used in earlier stages of GIC's development by an experimental garbage collector in order to allocate unwrapped LARs on the heap and also provided a convenient way for building the basic macros that provide access to a `TP_` structure's fields (`THE_VALS` , `THE_NESTED`).

However, these extra fields are not strictly necessary and therefore can be eliminated. As we have seen in Chapter 6, GIC currently uses an external garbage collector (Boehm GC) as the default. Also, all the required information for accessing the fields of a LAR (implemented as a `TP_` structure) is statically known.

Possible benefits from eliminating the extra fields include the following:

- The `TP_` structure becomes smaller (one word per LAR is saved) leading to smaller memory footprints for the compiled programs. As LARs are allocated on the heap in the presence of user-defined data types (and are in fact the only data objects that are ever allocated in memory), this also implies a reduced strain on the garbage collector. Some improvement of execution speed (gained indirectly, because of less garbage collection cycles during execution) might also be expected—this, however, is by no means to be taken for granted as it also depends on the particular garbage collector that is used and its configuration.
- One dereference, one offset addition, and one load operation can be replaced by a constant load operation in `THE_VALS` and `THE_NESTED` macros. A uniform speedup in the execution times of the compiled programs can be expected after this change.

- After the elimination of the extra fields it is possible to decouple the arities of the `the_args` and the `the_vars` arrays (see Figure 4.2) without adding another field in the `TP_` structure — decoupling these arities makes another optimization in the representation of LARs possible (see Subsection 7.1.4).

7.1.2 Implementation Outline

Although the target of removing the `arity` and `nesting` fields from the `TP_` structure is pretty clear and the lack of such fields in the description of lazy activation records in Section 4.1 can be seen as a possible indication of its feasibility, it is not immediately obvious that the specific C implementation LARs used in GIC admits this a change—at least in the case of the `arity` field.

More specifically, as long as the exact size of a LAR does not need to be determined dynamically, the `nesting` field can be immediately removed: as mentioned in Subsection 4.2.3, `nesting`, by itself, indicates the length of the list of contexts that correspond to nested `case` constructs, which is something of no interest. Therefore, when GIC compiles a program for running with the default garbage collector (currently Boehm GC, which implements its own mechanisms for keeping track of allocated objects' size) the `nesting` field is nowhere needed and can be safely removed.

Now, let's see why it is possible to also eliminate the `arity` field. We examine the cases where the fields of a LAR are accessed:

- A field of a LAR can be accessed in the body of the C function that accepts it as a parameter. As the form of the LAR that each function accepts is unique, its arity can be hard-coded in the function's body.
- A field of a LAR can be accessed in a function's body after an `ACTUAL` operation has been performed. `ACTUAL` occurs only in C functions that correspond to actual parameters in the input functional program. As every actual parameter occurs only once, the current context after the `ACTUAL` operation is always the same: it is the LAR of the unique function that creates (and uses in a call) this actual parameter. The arity of this LAR can, again, be hard-coded.
- The `the_nested` array of a LAR lar is accessed when a context corresponding to a `case` expression (i.e. the actual parameters of a constructor) is asked for. But this happens only when a specific actual parameter of a constructor is needed, i.e. when a field of another LAR lar' needs to be accessed. Fields of LARs representing data types (such as lar') are accessed through the *name* of a pattern-bound variable, say a. Keeping the name of the function enclosing the `case` construct that binds a suffices for knowing the arity of lar: lar is exactly the context of this function. This can be hard-coded in the body of the C function corresponding to the pattern-bound variable a.

The second case above may be better understood by looking at Example 4.2.1 and comparing the FOL program with the corresponding LAR representation. For the third case, recall that the names of pattern-bound variables are kept the same with the names of the formals of constructor-wrapping functions and therefore become separate C functions.

We should now look at Figure 4.2 again and consider what are the necessary changes. After removing the `arity` and `nesting` fields from the `TP_` structure (and from the anonymous structure to which the `LAR_STRUCT` macro expands), we must also eliminate them from the access-providing macros `THE_VALS` and `THE_NESTED`. As we now know that a LAR's arity is statically known wherever it is needed, we can pass this value directly to the macros as an extra argument wherever the macros are invoked. The same holds for the `GETARG` macro, which should propagate its new argument. We can also go a step further in advance, decoupling the arities of the `the_args` and the `the_vals`

arrays. The modified definitions described so far are presented in Figure 7.1 (should be compared with Figure 4.2).

Thin LARs in a separate compilation setting

It is clear that the optimization described in this section is orthogonal with separate compilation (as described in Section 3.2, separate compilation is implemented by the modular defunctionalization technique in GIC).

The key observation is that the arity of a LAR is statically known even if the LAR is used in a module other than the one where the function accepting it as an argument is defined: exactly because, as we explained earlier, we can always know which this function is.

7.1.3 Results

In order to evaluate the modification of GIC's LAR back-end described in Subsection 7.1.2, we compare the execution times of a small collection of benchmarks when compiled with the original LAR back-end and when compiled with the modified LAR back-end (where the `arity` and `nesting` fields of the `TP_` structure have been eliminated). Testing was performed on a machine with four quad-core Intel Xeon E7340 2.40GHz processors and 16 GB memory running Debian 6.0.7 (64-bit) with `gcc` version 4.4.5, `llvm-gcc` version 4.2.1 and `llvm` version 2.7. The results are summarized in Figure 7.2. For all tests garbage collection had been disabled in order to neutralize any effect coming from the garbage collector configuration and also to obtain the most deterministic behaviour possible during the execution of the benchmarks. However, the effect of lower memory consumption in the case of our modified back-end is certainly not completely isolated: for example, when we use the thinner implementation of LARs we could possibly gain execution speed also because of better data locality.

The table presents data for both external C compilers (namely, `gcc` and `llvm` through its `llvm-gcc` front end) that are currently used by GIC to compile the resulting C programs. However, the focus is on the behaviour of output programs when these are compiled with `llvm-gcc`. LLVM has been observed to consistently benefit the execution speed of programs compiled with GIC. Therefore, through the `llvm-gcc` front-end currently, LLVM is the primary target for GIC's LAR back-end—in the future, however, Clang could be used instead or, preferably, GIC could target LLVM directly.

As was more or less expected, it is evident that from the possible benefits described in Subsection 7.1.1, the second of them alone is significant enough: the average speedup of execution is almost 14% when we use `llvm-gcc` at the end of our compilation chain (which is also the case that interests us more) and approximately 5% when we combine GIC with `gcc`—note however that if we exclude the exceptional case of the "primes" program, where execution unexpectedly slows down when we compile with the modified back-end, the average speedup climbs to more than 7% for the case of `gcc`.

7.1.4 Conclusion

The results presented in the previous subsection indicate that the low-level optimization described earlier returned observable benefits in the execution speed of compiled programs. Looking at the course of implementing the optimization we may make the following observations:

- There is a good trade-off between the total effort and the derived benefits: the underlying idea is quite simple, no additional static analysis of any kind is required, and the technical issues that arose during development were mild and in most cases restricted only in the modules implementing GIC's C code generator.

```c
typedef unsigned char byte;

typedef struct T_* TP_;

typedef struct Susp {
  int constr;
  TP_ ctxt;
} Susp;

typedef Susp (*LarArg)(TP_);

typedef struct T_ {
  TP_ prev;              // link to parent LAR
  void* data[];          // the rest of this struct contains:
                         //   - array of args to evaluate (ARGS)
                         //   - computed thunk values (VALS)
                         //   - nested contexts (NESTED)
} T_;

#define LAR_STRUCT(n_arity_a, n_arity_v, n_nesting) \
  struct {                                          \
    TP_ prev;                                       \
    LarArg the_args[n_arity_a];                     \
    Susp the_vals[n_arity_v];                       \
    TP_ the_nested[n_nesting];                      \
  }

#define THE_ARGS(T)                         ((byte *) &((T)->data))
#define THE_VALS(VARSARITY, T)              \
    (THE_ARGS(T) + VARSARITY * sizeof(LarArg))
#define THE_NESTED(VARSARITY, VALSARITY, T) \
    (THE_VALS(VARSARITY, T) + VALSARITY * sizeof(Susp))

#define ARGS(x, T)                          (((LarArg*) THE_ARGS(T))[x])
#define VALS(x, VARSARITY, T)               \
    (((Susp*) THE_VALS(VARSARITY, T))[x])
#define NESTED(x, VARSARITY, VALSARITY, T)  \
    (((TP_*) THE_NESTED(VARSARITY, VALSARITY, T))[x])

#define VAR(x)        FUNC(x)
#define FUNC(x)       Susp x(TP_ T0)
#define ACTUAL        T0 = T0->prev

#define GETARG(x, ARGSARITY, T)  ({           \
      if (ARGS(x, T) != NULL) {               \
        Susp val = ARGS(x, T)(T);             \
        VALS(x, ARGSARITY, T) = val;          \
        ARGS(x, T) = NULL;                    \
      }                                       \
      VALS(x, ARGSARITY, T);                  \
    })

#define AR_S(n_arity_a, n_arity_v, n_nesting, ...)            \
  ((TP_) &((LAR_STRUCT(n_arity_a, n_arity_v, n_nesting))      \
    { T0, { __VA_ARGS__ } }))
```

Figure 7.1: Implementing LARs in C

Program	GIC (original back-end)		GIC (modified back-end)		Speedup (gcc)	Speedup (llvm-gcc)
	gcc	llvm-gcc	gcc	llvm-gcc		
ack	2.50	1.26	2.42	1.17	3.2%	12.7%
church	0.18	0.10	0.17	0.08	5.5%	11.1%
collatz	0.29	0.17	0.28	0.15	6.9%	11.8%
fib	1.36	1.25	1.14	0.95	16.2%	24.0%
ntak	8.60	5.84	8.19	4.86	4.8%	16.8%
primes	2.51	1.57	2.77	1.45	-10.4%	7.6%
queens-num	0.13	0.09	0.13	0.08	7.7%	11.1%

Figure 7.2: An evaluation of the modified back-end

- The changes do not seem to interfere with anything else in GIC at the moment and are also unlikely to pose any problems in the future: the low-level representation of LARs is irrelevant for most components of the compiler.
- The LAR intermediate representation, seen as a condensed description of C code, provides a useful thin layer which aids our understanding and makes the reasoning about C code generation easier.

The first two observations presumably suggest that another similar optimization could be successful: it is possible to have an even more compact C representation of LARs assuming that we use a strictness analysis at the level of FL. If we know that a function's argument is strict then we do not need to reserve space for it in the `the_args` array of the corresponding `TP_` structure, and if its value is demanded only once then we do not need to reserve space in the `the_vals` array.

7.2 Constant Applicative Form Memoization

This section describes the implementation of a top-level CAF memoization mechanism for GIC's LAR back-end. The back-end initially lacked such capability; however, lazy activation records proved to be a flexible and re-usable infrastructure in this case.

We start with a brief description of CAFs focusing on their role as compilation units and proceed with the presentation of the basic ideas of the implementation and the results.

7.2.1 Motivation

According to call-by-need semantics, a function argument is evaluated at most once. In the case of a practical full-blown lazy functional language this implies that every named expression that involves some computation must be evaluated at most once, i.e. all local and top-level definitions with a constant expression (which is not a lambda abstraction) as the right-hand-side must be evaluated at most once.

From the description of lazy activation records in Section 4.1 it becomes evident that the LAR back-end handles function arguments properly. Also, GIC's front-end uses a variant of lambda-lifting which eliminates local definitions in favor of top-level combinator definitions (see Chapter 8). Therefore, we will not be concerned with local definitions in the rest of this chapter. Top-level definitions however were not initially handled properly (i.e. in accordance with call-by-need semantics) by the LAR back-end: intuitively, even when the top-level definition of, say, name v is not a function in the FOL program (i.e. has arity zero) it will eventually be translated to a C function which will be called as many times

as the number of occurrences of v in the FOL program. Conforming with the semantics as well as better performance of compiled programs are good reasons for dealing with this issue.

To state the issue more clearly, we observe that in the absence of local definitions and anonymous functions we only need to handle top-level CAFs in a way that conforms with the call-by-need semantics. The following definitions are useful to also make clear what a CAF is in general and what it is in our specific case.

Definition 7.2.1 (combinator)**.** *A combinator is a lambda expression which contains no occurrences of a free variable [Bare84].*

Definition 7.2.2 (supercombinator)**.** *A supercombinator F is an expression of the form*

$$\lambda\, x_1.\, \lambda\, x_2.\, \ldots \lambda\, x_n.\, E \qquad (n \geq 0)$$

where (i) E is not a lambda abstraction, (ii) F has no free variables and (iii) every lambda abstraction in E is a supercombinator [Peyt87].

Definition 7.2.3 (CAF)**.** *A Constant Applicative Form (CAF) is a supercombinator of zero arity [Peyt87].*

In our particular case we only have top-level definitions in FOL and the body of such a definition cannot have any free variables. Therefore any definition in a FOL program with zero arity is a CAF.

7.2.2 Implementation Outline

The implementation should involve (i) the recognition of all top-level CAFs based on the arity of each definition; (ii) a memoization mechanism for the names defined as CAFs. But in fact there is already such a mechanism available: lazy activation records.

In the simplest case, all CAFs can be considered as the arguments of a single LAR, i.e. we can use a global `TP_` structure that stores pointers to C functions that correspond to CAFs in the FOL program. Subsequently, all calls to these functions in the C program are converted to requests for the values of the arguments of this LAR via the familiar `GETARG` macro.

Example 7.2.1. Consider the following simple FOL program

```
fib x = if x<2 then 1 else (fib (x-1)) + (fib (x-2))
f34 = fib 34
result = f34 + f34 + f34
```

where `f34` is obviously a CAF and occurs several times in the body of `result` —which, by the way, is also a CAF but this does not matter in our example. The C code corresponding to the definition of `result` is

```
FUNC(modMain_result){
  return (
    (Susp) { ( ((Susp) { ((__GCAF_MAIN(1)).constr +
                          (__GCAF_MAIN(1)).constr), NULL }).constr +
               (__GCAF_MAIN(1)).constr),
             NULL
           }
  );
}
```

and the missing part of the picture is the code for manipulating the global LAR:

```
#define __GCAF_MAIN(x)  GETARG(x, 3, __genv_Main)

#define __GCAF_MAIN_AR(arg1, arg2, arg3) ({         \
      TP_ lar = (TP_) GC_MALLOC(sizeof(T_) +        \
                                3 * sizeof(LarArg) + \
                                3 * sizeof(Susp) +   \
                                0 * sizeof(TP_));    \
      lar->prev = T0;                               \
      ARGS(0, lar) = arg1;                          \
      ARGS(1, lar) = arg2;                          \
      ARGS(2, lar) = arg3;                          \
      lar;                                          \
    })

static TP_ __genv_Main;

void __initModule_Main(TP_ T0) {
  __genv_Main = __GCAF_MAIN_AR(modMain_f34, modMain_result);
}
```

The global LAR's initializing function (`__initModule_Main`), which allocates memory for the LAR and sets its contents appropriately, is called in the `main` function at the beginning of execution.

Note that this simple approach has a significant shortcoming: the values resulting from the evaluation of CAFs are always reachable through the global LAR that stores them in its `the_vals` array and therefore are never garbage collected.

CAF memoization in a separate compilation setting

No serious problems arise when we combine the simple top-level CAF memoization scheme described in this subsection with separate compilation. The simplest approach is to create one global LAR per module that stores the CAFs of the present module. As the top-level function arities are saved (along with types and information needed by the Modular Defunctionalization technique) in interface files for every module, intermodular calls to CAFs can be detected during the compilation of a module and be redirected to the other module's global LAR.

7.2.3 Results and Conclusion

The implementation of the simple top-level CAF memoization mechanism works as expected and GIC's LAR back-end currently fully conforms with the call-by-need semantics. The expected execution time speedup can be observed in the case of programs that make non-trivial use of CAFs, such as the simple one presented in Example 7.2.1

The issue of (non-) garbage collection of CAFs however still remains. Although in most cases it does not pose any serious problems, there are cases where it does: for example, if the value of a CAF c is some big data structure and c is only used, say, once at the beginning of execution then the big data structure will survive the whole execution, possibly reserving a substantial amount of memory. Note that this was not a problem earlier when we did not store the values of CAFs anywhere: it is exactly the reference from the store (the global CAF) that prevents these data structures from being garbage collected.

Chapter 8

Lambda lifting

In this section we review lambda lifting, discuss its possible applications, and describe its role in GIC's compilation chain. Subsequently, we outline the implementation of the lambda lifter that was developed and integrated in GIC in the scope of the present dissertation, and also make some suggestions for the future development of this component.

8.1 Introduction

Lambda lifting is a program transformation which, in general, transforms an ordinary non-strict functional program (i.e. written in a language based on lambda calculus) into a set of (possibly mutually recursive) supercombinator definitions. In other words, lambda lifting eliminates all free variables from function bodies turning them into extra parameters. Usually, lambda abstractions are not included in the expressions of the target language and all functions are eventually turned into explicitly named top-level combinators. In general, free variables in function bodies introduce problems to the efficient execution of non-strict programs. Lambda lifting provides a solution to these inefficiencies and therefore it constitutes a compilation stage of many popular compilers. Lambda lifting may also eliminate the local definitions (i.e. `let` and `letrec` definitions), effectively "flattening" the input program's lexical structure and making it irrelevant for the rest of the compilation process.

It is an old fact (much older than the idea of lambda lifting) that lambda calculus can be translated into a CAF language, i.e. a language that only includes predefined (possibly higher order) constants and function application [Curr58]. Turner refines this approach in order to control the size of the output program [Turn79]. Moreover, his translation has the potentially important property that the evaluation of the output program corresponds to a *fully lazy evaluation* [1] of the source program. In general, his work reveals the opportunity of using such translations as realistic compilation techniques.

8.1.1 Translating lambda calculus into supercombinator definitions

Hughes observes that any combinator can be considered as a suitable operator for efficient execution and rejects predefined combinators in favor of compiler-generated supercombinator definitions. He originally uses a variant of lambda lifting to obtain such supercombinator definitions suitable for efficient execution in a graph reduction system [Hugh82]. Hughes' proposal still eliminates the problem

[1] An evaluation of a non-strict functional program is called *fully lazy* if every expression in the program is evaluated at most once after the variables in it have been bound [Hugh84]. Hughes makes an analogy with lazy evaluation, where every function argument is supposed to be evaluated at most once; however, lazy evaluation [Hend76] is a realization of the call-by-need evaluation strategy [Wads71], whereas fully lazy evaluation can be presumably better understood as a property of a particular lazy program—there is no corresponding implementation that can guarantee this property for an arbitrary functional program. From this perspective, a non-strict program can be translated to an extensionally equivalent one which (when lazy evaluation is used) has the property of full laziness.

of variables occurring free in functions while it also avoids some of the disadvantages associated with Turner's approach (when viewed from the perspective of compilation techniques).

Example 8.1.1. Intuitively, given a lambda expression $e \doteq \lambda\, x.\, E$, we can perform lambda lifting taking the following steps:

1. Deal with the function's body recursively. This step yields a term $e' \doteq \lambda\, x.\, E'$, where E' is an applicative form, i.e. only contains applications of combinators and constants.
2. For each free variable occurring in e' abstract the term so that the variable becomes bound. This step yields a term of the form $\lambda\, a_1.\ \ldots \lambda\, a_k.\, e'$ (call this term e''), where $a_1, \ldots, a_k$ are the variables that appear free in e'.
3. Term e'' from the previous step is a combinator; we can give it a name c, add its definition $c\ a_1\ \ldots\ a_k\ x = E'$ at top-level, and replace e'' with the application $c\ a_1\ \ldots\ a_k$.

Consider the following program written in Haskell syntax:

```
g n     = (\f → (\f' → (f' 1) * (f' 8)) (f (f n 4)))
          (\x y → x * x * n + y)
result  = g 5
```

According to the rules described earlier, variable `n` must be abstracted from both terms in the application in the body of `g` and the resulting supercombinators must be named and defined at top-level. The inner expression `\f' → (f' 1) * (f' 8)` is already a supercombinator. Here is the resulting program:

```
c1 n x y = x * x * n + y
c2 f'    = (f' 1) * (f' 8)
c3 n f   = c2 (f (f n 4))
g n      = (c3 n) (c1 n)
result   = g 5
```

Here we have actually eliminated an extra trivial combinator `\n x → c1 n` (by η-reducing it on-the-fly) which arises by treating the lambda abstraction `\x y → x * x * n + y` strictly according to the aformentioned rules.

Having efficient compilation in mind, Hughes describes several optimizations that could be combined with lambda lifting and could lead to faster execution of the generated programs, such as generating programs that possess the property of fully lazy evaluation, supercombinator parameter ordering, conditional optimization, etc. [Hugh82, Hugh84]. While most of these optimizations can be considered as "mid-level" compilation techniques targeting at eliminating redundancy, fully lazy evaluation could be worth to perform in the front-end regardless of the rest of the design of a particular compiler: given a runtime environment that supports laziness (and regardless of the exact mechanism that is employed), programs possessing the property of fully lazy evaluation should always retain the advantage of never repeating computations. Hughes describes a way of performing lambda lifting so that the resulting program always has the property of fully lazy evaluation. The idea is that instead of abstracting free variables (in the manner we saw in Example 8.1.1), which are the *minimal* free subexpressions of the given lambda expression, we can detect (Hughes describes a way for doing so) and subsequently abstract the *maximal* free subexpressions [2].

[2] The free subexpression of a given expression e are these subexpressions that do not contain occurrences of bound variables the binding abstraction of which also resides in e. The maximal free subexpressions are these that are not part of any larger free subexpression.

Example 8.1.2. Let's consider again the program from Example 8.1.1:

```
g n      = (\f → (\f' → (f' 1) * (f' 8)) (f (f n 4)))
           (\x y → x * x * n + y)
result   = g 5
```

The execution of this program is not fully lazy. In particular, we can see that the subexpression `x * x * n` will be computed more than once after its variables have been bound: one time during the evaluation of `f' 1` and one more during the evaluation of `f' 2`. The same holds for the lambda-lifted program presented in Example 8.1.1.

Now, let's follow Hughes' approach and identify the maximal free subexpressions. We can see that variable `n` is still a maximal free subexpression of the two abstractions in the application in the body of `g`. However, for the inner lambda abstraction `\y → x * x * n + y` the whole expression `x * x * n` is a maximal free subexpression, and this is what we should abstract instead of just `n`. The resulting program is the following:

```
c1 m y   = m + y
c1' n x  = c1 (x * x * n)
c2 f'    = (f' 1) * (f' 8)
c3 n f   = c2 (f (f n 4))
g n      = (c3 n) (c1' n)
result   = g 5
```

Observe that in this case the subexpression `x * x * n` will be evaluated only once after the variables in it have been bound. In fact, the same holds for every subexpression of this program and, therefore, its evaluation is fully lazy. Notice that combinator `c1'` is not trivial in this case as it was in Example 8.1.1.

8.1.2 Variants and implementations of lambda lifting

There are several possible ad hoc ways to perform lambda lifting and a few different concrete algorithms. Lifting free variables in function bodies has been proved correct with respect to denotational semantics by Danvy [Danv99]. Fischbach and Hannan propose a general specification for lambda lifting, that is not bound to any particular lambda lifting algorithm, and prove it correct with respect to simple typing and call-by-name operational semantics [Fisc00]. Subsequently, as an example, they use their specification to prove the correctness of a concrete algorithm for lambda lifting. For the rest of this chapter we will focus only on a some concrete algorithms, as our particular purpose is to finally come up with a working solution for GIC.

Johnsson independently develops lambda lifting and uses it in combination with his technique for compiling supercombinator definitions to G machine code [John84]—this whole work was also part of the implementation of a compiler for Lazy ML [Augu84]. He describes a lambda lifting algorithm for transforming functional programs to recursive equations (i.e. top-level, possibly mutually recursive, supercombinator definitions) that was designed with this particular setting in mind [John85]. While Hughes is not concerned with local definitions (he desugars local non-recursive definitions to lambda expressions, and local recursive definitions to the application of the Y fix-point combinator), Johnsson rejects this treatment in the context of his purpose: he notes, among other things, that the Y combinator introduces unnecessary inefficiencies for his implementation and that it is also unnecessary to abstract *all* free variables that occur inside a lambda expression. The latter observation is valid even in the case where local definitions were not to be eliminated and, in all cases, abstracting all free variables in lambda expressions does not retain direct recursion; it leads to indirect recursive calls on function arguments instead. A key idea behind Johnsson's algorithms is to treat local definitions (i.e. `let` and

`letrec` definitions) specially: function names do need not be abstracted out, as the (possibly local) definitions defining them will eventually end up at top level.

Example 8.1.3. A slightly simplified version of Johnsson's algorithm for lambda lifting can be roughly described as follows:

1. Give all identifiers a unique name.
2. Give all anonymous functions a unique name introducing a local definition for each anonymous function.
3. Compute the set of variables to be abstracted out of each function defined in the program. Essentially, we need to compute the transitive closure C^* of the relation $C \subseteq (\text{defined names}) \times (\text{defined names})$ where $C = \{(f, g) \mid g \text{ occurs in the body of } f\}$. Note that function names include top-level and locally defined names. Now, if E_f denotes the set of variables to be abstracted out of the definition of f, S_g the set of (lambda bound) variables occurring free in g, and $X = \{h \mid (f, h) \in C^*\}$ then $E_f = \bigcup_{g \in X} S_g$.
4. For each definition $f_i = e_i$ we abstract all variables in E_{f_i} out of e_i and substitute each occurrence of f_i in the program for the application of f_i on the extra arguments.
5. All definitions are lifted at top-level.

Let's consider for one more time the program from Example 8.1.1. Applying Johnsson's algorithm on this yields the same lambda lifted program as the one obtained in Example 8.1.1 by following the approach of Hughes. However, Johnsson's algorithm can take advantage of a more natural equivalent presentation of our input program:

```
g n      = let f  = \x y → x * x * n + y
               f' = f (f n 4)
           in  f' 1 * f' 8
result   = g 5
```

Notice that all functions already have unique names that the only variable that needs to be abstracted out of the definitions of `f` and `f'` is `n`. Following the 4^{th} step described above, we abstract `n` as necessary and amend all occurrence of `f` and `f'` to receive the extra argument:

```
let f  = \n x y → x * x * n + y
    f' = \n → f n (f n n 4)
in  f' n 1 * f' n 8
```

We can now bring the combinator definitions at top-level and eliminate the `let` construct. Here is the resulting program:

```
f n x y = x * x * n + y
f' n    = f n (f n n 4)
g n     = f' n 1 * f' n 8
result  = g 5
```

Notice that the output program above does not possess the property of fully lazy evaluation: the subexpression `x * x * n` will be evaluated more than once after all variables occurring in it have been bound.

From the informal description of Johnsson's algorithm given in Example 8.1.3, the 4^{th} step is the most costly: its worst case asymptotic time complexity $\mathcal{O}(m^3)$, where m is the number of definitions

in the program. Johnsson describes an equivalent way of obtaining the sets E_{f_i} as the solution of a set of recursive equations [John85]. The equations are constructed and gradually solved during a top-down traversal of the input program. The total time complexity of Johnsson's algorithm is $\mathcal{O}(n^3)$, where n is the size of the program. Danvy and Schultz improve the performance of this approach to $\mathcal{O}(n^2)$—which is asymptotically optimal for lambda lifting—by using the input program's call graph to group together the functions that need the same extra parameters [Danv04]. Morazán and Schultz also propose a graph-based approach running in $\mathcal{O}(n^2)$ which is also optimal with respect to the set of variables to be abstracted out of definitions [Mora08]—Johnsson's original algorithm is optimal from this point of view.

Peyton Jones and Lester seperate the concepts of lambda lifting and full laziness and show that the these processes can be performed as two independent transformations [Jone91]. They use a simple functional language with local definitions as the source and target language and they provide their own (modular) implementation of fully lazy lambda lifting, where the full laziness property for the output programs is obtained with the initial application of an independent transformation involving `let` expressions. The core idea is the same one that Hughes describes, that is to abstract out maximal free subexressions; here, however, it is implemented by naming maximal free subexpressions using local definitions, and subsequently "floating" these definitions outwards as much as possible. The programs generated by this transformation possess the fully lazy evaluation property regardless of whether a second lambda-lifting phase follows or not. Therefore, any algorithm for lambda lifting could be used afterward; in fact, even implementations that do not perform lambda lifting at all could benefit. In their implementation Peyton Jones and Lester use a lambda lifter that retains local definitions (as they are supported in the target language) but otherwise follows Hughes' approach. As we mentioned earlier, this approach has the potential disadvantage of not retaining direct recursion.

Peyton Jones and Lester describe in detail and implement a method for detecting maximal free subexpressions; they suggest though that abstracting out *all* maximal free subexpressions could lead to inefficiencies too: we can end up with more supercombinator definitions (and small execution steps, respectively) and discourage some compiler optimizations by removing subexpressions from their context. Instead, they propose the use of some heuristics in order to decide whether to lift an expression or not. For example, there is little point in lifting a constant or a non-reducible expression.

8.2 A lambda lifter for GIC

In this section we shall discuss the needs of GIC with respect to lambda lifting, argue that Johnsson's original approach fits our purposes, and justify the choice to follow this approach. Finally, we address the issue of full laziness in the context of our compiler in particular.

8.2.1 Options and restrictions

The need for lambda lifting in our case comes from the fact that the intensional transformation that is used in GIC (presented in Section 2.2) does not support local definitions nor anonymous functions. This already excludes lambda lifting variants (such as the one proposed by Peyton Jones and Lester, see Subsection 8.1.2) that do not eliminate local definitions. Using the Y combinator for recursive local definitions would be inefficient under our compilation scheme which translates each top-level first order combinator to a C function—a FOL (Figure 2.4) program is in fact a set of first-order combinators. Johnsson's approach, however, seems to fit well in our context: (i) it names all anonymous functions and eliminates all local definitions, (ii) avoids the usage of the Y combinator, (iii) also retains direct recursion (which results in directly recursive C functions instead of indirectly recursive calls to function pointers residing in a LAR), and (iv) abstracts out of each definition the minimum

number of variables and takes advantage of the initial lexical structure of the input program in order to avoid breaking the execution into many small steps. These properties give Johnsson's approach a decisive advantage in our case.

In general, a lambda lifter seems to be an easy addition in GIC. There seems to be no particular problem (at least from the point of view of semantics) with performing defunctionalization after lambda lifting—this is a composition of two meaning-preserving transformations which should give a meaning-preserving transformation. Also, lambda lifting does not affect the types of functions defined at top-level in the input program. This makes lambda lifting easily applicable in a separate compilation setting: it suffices to lambda lift each module separately.

8.2.2 Integrating a Johnsson-style lambda lifter in GIC

From the discussion in the previous subsection it is clear that Johnsson's algorithm is currently the most appropriate for our case. It serves our initial purpose avoiding also inefficiencies at the same time. Moreover, there is no need for any modifications, the algorithm fits in as it is—note that, in contrast with the need to eliminate the `let` and `letrec` constructs, lambda lifting does not need to be concerned with the `case` constructs, as these are supported by the intensional transformation.

The lambda lifter that was developed for GIC essentially implements the original algorithm proposed by Johnsson, and also follows his approach for constructing and solving the recursive equations that describe the sets of variables to be abstracted out of each definition.

8.2.3 Observations and suggestions

The properties of our compiler's design essentially pointed at the solution we were looking for. Certainly, the adoption of an existing, clearly presented, and much used in practice algorithm generally increases the confidence in the correctness of our design. And also in practice, our lambda lifter works as expected and fulfills our major objective, that is to eliminate the local definitions, effectively allowing us to extend GIC's source language towards real Haskell.

However, there are some issues related with lambda lifting that have not been addressed yet. In general, interactions between the intensional transformation and any other program transformation operating at the FL level have not yet been studied in detail—the fact that the intensional transformation preserves the semantics tells us nothing about such interactions. Moreover, the presence of defunctionalization in our compilation chain could also affect decisions on the design of other source level transformations.

Full laziness

In the case of lambda lifting in particular, the first choice between (roughly) a Johnsson-style and a Hughes-style approach was supported by some strong benefits in the case of the former approach. However, if lambda lifting is to be combined with compilation for full laziness, then things get more complicated: firstly, we should verify that our implementation could benefit from such an approach, i.e. that the "extra laziness" passes through defunctionalization and the intensional transformation and is reflected in the generated C program.

Example 8.2.1. Defunctionalizing the lambda lifted program from Example 8.1.2 (which, as explained earlier, has the property of fully lazy evaluation) yields the following program:

```
data Closure_IIII     = C11
data Closure_III      = C1 | C11_I Int
data Closure_ILIIIRI  = C3
```

```
data Closure_LIIIRI   = C3_I Int
data Closure_II       = C1_I Int | C11_I_I Int Int

apply_II_I f a =
  case f of
    C11_I_I n1 n2 → apply_II_I (c11 n1 n2) a
    C1_I n1 → c1 n1 a

apply_III_I f a =
  case f of
    C1 → C1_I a
    C11_I n → c11 n a

apply_IIII_I f a =
  case f of
    C11 → C11_I a

apply_ILIIIRI_I f a =
  case f of
    C3 → C3_I a

apply_LIIIRI_LIIIR f a =
  case f of
    C3_I n → c3 n a

c1 m y  = m + y
c11 n x = apply_III_I C1 (x * x * n)
c2 f1   = (apply_II_I f1 1) * (apply_II_I f1 8)
c3 n f  = c2 (apply_III_I f (apply_II_I (apply_III_I f n) 4))
g n     = apply_LIIIRI_LIIIR (apply_ILIIIRI_I C3 n) (apply_IIII_I C11 n)
result  = g 5
```

As might have been expected, we can see that in the defunctionalized program the subexpression `x * x * n` is, again, evaluated exactly once after its variables have been bound, i.e. the extra laziness is retained. The same holds for the intensional program derived by the application of the intensional transformation on the defunctionalized program above:

```
data Closure_IIII     = C11
data Closure_III      = C1 | C11_I Int
data Closure_ILIIIRI  = C3
data Closure_LIIIRI   = C3_I Int
data Closure_II       = C1_I Int| C11_I_I Int Int

apply_II_I =
  case apply_II_I_f of
    C11_I_I -> call[0] (apply_II_I)
    C1_I -> call[0] (c1)

apply_III_I =
  case apply_III_I_f of
    C1 -> call[0] (C1_I)
    C11_I -> call[1] (c11)

apply_IIII_I =
  case apply_IIII_I_f of
    C11 -> call[0] (C11_I)

apply_ILIIIRI_I =
```

```
  case apply_ILIIIRI_I_f of
    C3 -> call[0] (C3_I)

apply_LIIIRI_LIIIR =
  case apply_LIIIRI_LIIIR_f of
    C3_I -> call[0] (c3)

c1      = c1_m + c1_y
c11     = call[0] (apply_III_I)
c2      = call[1] (apply_II_I) * call[2] (apply_II_I)
c3      = call[0] (c2)
g       = call[0] (apply_LIIIRI_LIIIR)
result  = call[0] (g)

C11     = C11
C1      = C1
C11_I   = C11_I
C3      = C3
C3_I    = C3_I
C1_I    = C1_I
C11_I_I = C11_I_I

C11_I_0   = actuals[apply_IIII_I_a]
C11_I_I_0 = actuals[]
C11_I_I_1 = actuals[]
C1_I_0    = actuals[apply_III_I_a]
C3_I_0    = actuals[apply_ILIIIRI_I_a]
apply_IIII_I_f  = actuals[call[0] (C11)]
apply_IIII_I_a  = actuals[g_n]
apply_III_I_f   = actuals[call[0] (C1), c3_f, c3_f]
apply_III_I_a   = actuals[(c11_x * c11_x) * c11_n,
                          call[3] (apply_II_I), c3_n]
apply_II_I_f    = actuals[call[0] (c11), c2_f1, c2_f1,
                          call[2] (apply_III_I)]
apply_II_I_a    = actuals[apply_II_I_a, 1, 8, 4]
apply_ILIIIRI_I_f    = actuals[call[0] (C3)]
apply_ILIIIRI_I_a    = actuals[g_n]
apply_LIIIRI_LIIIR_f = actuals[call[0] (apply_ILIIIRI_I)]
apply_LIIIRI_LIIIR_a = actuals[call[0] (apply_IIII_I)]
c1_m  = actuals[C1_I_0]
c1_y  = actuals[apply_II_I_a]
c11_n = actuals[C11_I_I_0, C11_I_0]
c11_x = actuals[C11_I_I_1, apply_III_I_a]
c2_f1 = actuals[call[1] (apply_III_I)]
c3_n  = actuals[C3_I_0]
c3_f  = actuals[apply_LIIIRI_LIIIR_a]
g_n   = actuals[5]
```

So, it seems possible that the property of full laziness is retained by the composition of defunctionalization and the intensional transformation. More precisely, the desired property would be that if, after lambda lifting, there is an expression e in an FL program p that is evaluated at most once during the evaluation of p, then this expression is also evaluated at most once during the evaluation of the corresponding intensional program p' obtained by the application of the intensional transformation on p. However, currently there is no formal proof of this claim.

Supposing that it is indeed the case that our compilation scheme can benefit from a transformation operating at the FL level that "adds laziness" to the compiled program, we can follow the approach

of Peyton Jones and Lester described earlier and combine a separate "front-end" component with our existing Johnsson-style lifter. But the next question would have to do with the details of this transformation: as explained in Subsection 8.1.2, it is possible that, in practice, *full laziness* is not exactly what a compiler designer would wish for. There are some heuristics that can aid the compiler in deciding whether extra laziness is *usefull* or not at some point, but these have been proposed on the basis of the lazy evaluation model of execution and, usually, with some graph-reduction derived underlying implementation in mind. In our particular case, however, defunctionalization and the intensional transformation (and although both semantics-preserving) must be taken into account, along with the design of the runtime environment (lazy activation records), in order to reach definite conclusions on the effects of extra laziness on the execution time and memory consumption of the compiled programs.

As a first step, in order to get a better intuition on issues related with lambda lifting, we have developed a separate, flexible prototype implementation of lambda lifting and the full-laziness transformation. It is possible that the capability to gain laziness (full laziness in the strict sense does not seem to be particularly favorable as explained earlier) will soon be added to GIC, perhaps via some optimization switch.

Faster lambda lifting for faster compilation

As mentioned in Subsection 8.1.2, Johnsson's original lambda lifting transformation uses a relatively expensive (in terms of asymptotic time complexity) algorithm—at least for the context of a compiler. In practice the algorithm should perform much better on average; however there has been no extensive testing focused on the operation of the lambda lifter so far. In any case, and especially if long compile times are observed during testing GIC with real programs, it could be worthwhile to implement the proposal of Danvy and Schultz [Danv04] that leads to quadratic time (i.e. optimal time) lambda lifting.

Chapter 9

Conclusion

In this chapter we summarize the contributions of the present dissertation and make several suggestions regarding possible future work related with topics that were addressed (one way or another) in all previous chapters.

9.1 Contribution

The tangible contribution of this work includes several modifications of existing code as well as the implementations of some new small components in the GIC project. More specifically, within the scope of this dissertation the following actions regarding the GIC project were taken:

- The C code generator of GIC's LAR back-end has been modified in order to eliminate the `arity` and `nesting` fields from the `TP_` (i.e. LAR) structure. This effort resulted in a moderate increase in the execution speed of the compiled programs. For further details see Chapter 7.
- A simple but necessary CAF memoization mechanism has been developed in order to closer follow the Haskell specification. In marginal cases programs can practically benefit from this mechanism and gain execution speed. For further details see Chapter 7.
- A lambda lifter has been developed from scratch and integrated in GIC's front-end. This component currently allows GIC to handle a larger subset of Haskell. Moreover, and although not strictly necessary to do so, the optimization for full laziness (as described by Peyton Jones and Lester [Jone91]) can be combined with this lambda lifter. For further details see Chapter 8.
- Some minor modification were made in the C code generator in order to support the integration of the generated programs with an implementation of the Boehm-Demers-Weiser garbage collector. This effort added the missing feature of reliable garbage collection to our implementation and made it possible to test GIC with some more complex and demanding benchmarks. For further details see Chapter 6.
- A few minor bugs in other parts of the compiler were discovered and fixed.

Note that no solution was known in advance: after detecting an occasional problem, most time was actually spent in searching and studying existing work and subsequently comparing the possible solutions, rather than in implementing the solution that finally turned out to be the best one. The current dissertation could also be used as a kind of documentation on (or short guide for) GIC's current internal design. Finally, many chapters include short and condensed reviews of large parts of the existing literature on topics that are otherwise rather wide; these reviews could be a good starting point for getting a first impression and quickly becoming familiar with these topics—especially if compiler implementation is the motivation.

9.2 Future Work

There is certainly much more work to be done on the development of the GIC compiler towards turning this project to a full-fledged, competitive Haskell compiler. Parts of this possible future work, however, can also be considered out of our project's context, and constitute interesting independent research topics.

- Some proposals for future work associated with GIC's front-end are the following:
  - In the current implementation of GIC the FL intermediate language is simply typed. If GIC is to support Haskell in full at some point, FL should be combined with a type system that supports parametric polymorphism—note that although justified by the Haskell 98 specification, adopting ML-style polymorphism for FL would probably complicate things. As we saw in Section 3.2, the original modular defunctionalization described in [Four13a], which is the defunctionalization variant currently used in GIC, uses simply typed source and target languages. Therefore, this technique should be also adapted to cope with parametric polymorphism. A forthcoming paper addresses this issue and presents a modified modular defunctionalization transformation that also handles polymorphism following the approach of Gothier and Pottier [Pott06].
  - At some point, the issue of fully lazy evaluation of the generated programs arose, and its possible impact was intuitively estimated. As there seem to be possible benefits from this property in our setting, a more thorough investigation would be useful before moving forward to extend the implementation of GIC. It seems that the most important step for reaching any conclusions is to establish, for an arbitrary FOL expression e, a formal relationship between the number of execution steps (in accordance with the small-step operational semantics of FOL) involved in the evaluation of e and the number of execution steps involved in the evaluation of the corresponding intensional expression e' under the eductive evaluation model of execution of NVIL. The results of such an investigation could immediately be used in practice to decide on the importance of adding laziness at the FL level and, possibly, of other source-level transformations too.
  - It is apparent that GIC would benefit from a full-fledged front-end: our current primitive front-end does not perform any optimizations at all, effectively putting the whole strain for good performance to the LAR back-end. A mature, optimizing front-end, however, would almost certainly highlight our compiler's capabilities and would make possible a direct but nonetheless fair comparison between GIC and some popular Haskell compilers. Moreover, using an external front-end would also probably spare much time and development effort towards the goal of fully supporting the whole Haskell language. Our FL language (extended with parametric polymorphism as described earlier) would be possibly a good interface between an external Haskell front-end and GIC. A similar option would be to turn GIC into a back-end of a popular Haskell compiler (possibly GHC) instead of just adding to it a stand-alone front-end. This solution would presumably carry the extra benefit that GIC would have a chance to quickly become popular and that much more people would test and strain the implementation.
- Some proposals associated with GIC's LAR back-end are the following:
  - Our basic compilation strategy, that passes through the intensional transformation, seems to generate efficient code that is also characterized by a high degree of regularity. However, as explained in Chapter 5, there many low-level details over which we do not have any control at the C level (assuming that we also desire to retain portability), but which have been demonstrated to affect the performance of the compiled programs. Adding a more low-level code generator to the LAR back-end would potentially benefit performance, making GIC

more competitive. The portability concerns seem to favor an LLVM code generator in particular. However, as explained in Subsection 6.3.4, the issue of garbage collection could also affect the decision on the code generator. On the one hand, a native code generator seems to be better for accurate garbage collection. On the other hand, accurate garbage collection is still possible in the case of an LLVM code generator: shadow stack based approaches as well as the generation of stack maps are supported by LLVM [LLVM]. However, as mentioned in Subsection 6.3.1, there are some performance considerations in the case of a portable shadow-stack based implementation. Using stack maps should be very efficient, but it is not fully portable, and seems to be rather tricky. In any case, it seems that there is some strain between accurate garbage collection and portability; therefore their importance should be reconsidered carefully and with the knowledge that putting these in some order influences the possible options for the code generator.

– In Section 7.1 we discussed a low-level optimization in the C code generator dismissing the redundant fields `arity` and `nested` from the `TP_` structure, saving also some pointer arithmetic when a LAR is accessed, and ultimately resulting in better execution speed. However, there is more to be done in this direction, i.e. we could make LARs even "thinner": for example, a function argument that is known to be strict could be evaluated just before the function call and store its value directly in a slot in the `the_vals` array. We actually do not need to keep a slot in the `the_args` array for this argument. Symmetrically, if a function argument is known to be used only once then we do not need to save the value resulting from its evaluation. Therefore, we do not need to keep a slot in the `the_vals` array for this argument. In order to obtain the information on whether a function is strict or not on a particular argument some kind of strictness analysis is needed; however, even a simple form of such analysis would probably suffice for gaining some execution speed.
– Currently, all C functions in the generated program return a `Susp` structure as their result which, in general, represents a tuple containing a data constructor and a context holding its arguments. In the case of a return value of ground type, a degenerate `Susp` structure is actually returned, containing the ground return value in its (word-sized) first field and, by convention, a `NULL` pointer in its second field. A useful low-level optimization would be to avoid constructing and returning the degenerate `Susp` structure in the latter case and return just the unwrapped ground value instead. This would save a word in the stack per activation record, the initialization of the second field of the `Susp` structure with `NULL`, and the indirect access to its first field to take the actual return value. Moreover, the C compiler would presumably perform more easily a further optimization to return the ground value in a register instead of using the stack.

Bibliography

[Appe87] Andrew W. Appel, "Garbage Collection Can Be Faster Than Stack Allocation", 1987.

[Appe88] Andrew W. Appel, "Simple generational garbage collection and fast allocation", Technical Report CS-TR-143-88, Princeton University (NJ US), 1988.

[Appe89] Andrew W. Appel, "Runtime Tags Aren't Necessary.", *Lisp and Symbolic Computation*, vol. 2, no. 2, pp. 153–162, 1989.

[Arvi90] K. Arvind and Rishiyur S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture", *IEEE Trans. Comput.*, vol. 39, no. 3, pp. 300–318, March 1990.

[Ashc77] Edward A. Ashcroft and William W. Wadge, "Lucid, a Nonprocedural Language with Iteration", *Communications of the ACM*, vol. 20, no. 7, pp. 519–526, 1977.

[Ashc85] Edward A. Ashcroft and William W. Wadge, *Lucid, the dataflow programming language*, no. 22 in A.P.I.C. studies in data processing, Academic Press, 1985.

[Augu84] Lennart Augustsson, "A compiler for lazy ML", in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pp. 218–227, New York, NY, USA, 1984, ACM.

[Bake92] Henry G. Baker, "The treadmill: real-time garbage collection without motion sickness", *SIGPLAN Not.*, vol. 27, no. 3, pp. 66–70, March 1992.

[Bare84] Hendrik Pieter Barendregt, *The Lambda Calculus – Its Syntax and Semantics*, vol. 103 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, 1984.

[Bart88] Joel F. Bartlett, "Compacting garbage collection with ambiguous roots", *SIGPLAN Lisp Pointers*, vol. 1, no. 6, pp. 3–12, April 1988.

[Bart89a] Joel F. Bartlett, "Mostly-Copying Garbage Collection Picks Up Generations and C++", Technical Note TN-12, Digital Western Research Laboratory, Palo Alto, CA, USA, October 1989.

[Bart89b] Joel F. Bartlett, "SCHEME->C: a Portable Scheme-to-C Compiler", Technical report, WRL Research Report 89/1, 1989.

[Bell97] Jeffrey M. Bell, Françoise Bellegarde and James Hook, "Type-driven Defunctionalization", in *In Proc. 2^{nd} International Conference on Functional Programming*, pp. 25–37, ACM, 1997.

[Boeh] Hans-Juergen Boehm, "A garbage collector for C and C++", `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`. Accessed: 2013 August 25.

[Boeh88] Hans-Juergen Boehm and Mark Weiser, "Garbage collection in an uncooperative environment", *Softw. Pract. Exper.*, vol. 18, no. 9, pp. 807–820, September 1988.

[Boeh91] Hans-J. Boehm, Alan J. Demers and Scott Shenker, "Mostly parallel garbage collection", in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pp. 157–164, New York, NY, USA, 1991, ACM.

[Boeh92] Hans-J. Boehm, "A Proposal for Garbage-Collector-Safe C Compilation", *The Journal of C Language Translation*, vol. 4, no. 2, pp. 126–141, December 1992.

[Boeh93] Hans-Juergen Boehm, "Space efficient conservative garbage collection", *SIGPLAN Not.*, vol. 28, no. 6, pp. 197–206, June 1993.

[Boeh96] Hans-J. Boehm, "Simple garbage-collector-safety", in *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pp. 89–98, New York, NY, USA, 1996, ACM.

[Boeh02] Hans-J. Boehm, "Bounding space usage of conservative garbage collectors", *SIGPLAN Not.*, vol. 37, no. 1, pp. 93–100, January 2002.

[Boqu99] Urban Boquist, *Code Optimization Techniques for Lazy Functional Languages*, Ph.D. thesis, Chalmers University of Technology, Göteborg University, March 1999.

[Cejt00] Henry Cejtin, Suresh Jagannathan and Stephen Weeks, "Flow-Directed Closure Conversion for Typed Languages", in *In ESOP '00 [ESOP00*, pp. 56–71, Springer-Verlag, 2000.

[Char08] Angelos Charalambidis, Athanasios Grivas, Nikolaos S. Papaspyrou and Panos Rondogiannis, "Efficient Intensional Implementation for Lazy Functional Languages", *Mathematics in Computer Science*, vol. 2, no. 1, pp. 123–141, 2008.

[Chen70] C. J. Cheney, "A nonrecursive list compacting algorithm", *Commun. ACM*, vol. 13, no. 11, pp. 677–678, November 1970.

[Coll60] George E. Collins, "A method for overlapping and erasure of lists", *Commun. ACM*, vol. 3, no. 12, pp. 655–657, December 1960.

[Curr58] H.B. Curry and R. Feys, *Combinatory Logic*, no. τ. 1 in Combinatory Logic, North-Holland Publishing Company, 1958.

[Danv99] Olivier Danvy, "An Extensional Characterization of Lambda-Lifting and Lambda-Dropping", in *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, FLOPS '99, pp. 241–250, London, UK, UK, 1999, Springer-Verlag.

[Danv01] Olivier Danvy and Lasse R. Nielsen, "Defunctionalization at Work", in *Proceedings of the 3^{rd} ACM SIGPLAN international conference on principles and practice of declarative programming*, pp. 162–174, ACM, 2001.

[Danv04] Olivier Danvy and Ulrik Pagh Schultz, "Lambda-Lifting in Quadratic Time", *Journal of Functional and Logic Programming*, vol. 2004, 2004.

[Deme90] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow and Scott Shenker, "Combining generational and conservative garbage collection: framework and implementations", in *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pp. 261–269, New York, NY, USA, 1990, ACM.

[Detl90] David L. Detlefs, "Concurrent, atomic garbage collection", Technical Report CMU-CS-90-177, Carnegie-Mellon Univ. Computer Science Dept., Pittsburgh, Pennsylvania, October 1990. PhD thesis.

[Edel92] Daniel R. Edelson, "A mark-and-sweep collector for C++", in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pp. 51–58, New York, NY, USA, 1992, ACM.

[Faus87] Anthony A. Faustini and William W. Wagde, "An eductive interpreter for the language pLucid", in *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pp. 86–91, New York, NY, USA, July 1987, ACM.

[Feni69] Robert R. Fenichel and Jerome C. Yochelson, "A LISP garbage-collector for virtual-memory computer systems", *Commun. ACM*, vol. 12, no. 11, pp. 611–612, November 1969.

[Fisc00] Adam Fischbach and John Hannan, "Specification and Correctness of Lambda Lifting", in *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, SAIG '00, pp. 108–128, London, UK, UK, 2000, Springer-Verlag.

[Four11] Georgios Fourtounis, Nikolaos Papaspyrou and Panos Rondogiannis, "The Intensional Transformation for Functional Languages with User-Defined Data Types", in *Proceedings of the 8th Panhellenic Logic Symposium*, pp. 38–42, 2011.

[Four13a] Georgios Fourtounis and Nikolaos Papaspyrou, "Supporting Separate Compilation in a Defunctionalization-based Compiler", in *Proceedings of the* 2^{nd} *International Symposium on Languages, Applications and Technologies*, 2013.

[Four13b] Georgios Fourtounis, Nikolaos Papaspyrou and Panos Rondogiannis, "The Generalized Intensional Transformation for Implementing Lazy Functional Languages", in *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL '13)*, Rome, Italy, January 2013, ACM. To appear.

[Gold91] Benjamin Goldberg, "Tag-free garbage collection for strongly typed programming languages", *SIGPLAN Not.*, vol. 26, no. 6, pp. 165–176, May 1991.

[Griv04] A. Grivas, "Implementation of Functional Languages using the Branching Dimensions Transformation", Master's thesis, National Technical University of Athens, October 2004.

[Hast91] Reed Hastings and Bob Joyce, "Purify: Fast detection of memory leaks and access errors", in *In Proc. of the Winter 1992 USENIX Conference*, pp. 125–138, 1991.

[Hend76] Peter Henderson and James H. Morris, Jr., "A lazy evaluator", in *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pp. 95–103, New York, NY, USA, 1976, ACM.

[Hend02] Fergus Henderson, "Accurate Garbage Collection in an Uncooperative Environment", in *In Proceedings of the Third International Symposium on Memory Management*, pp. 150–156, ACM Press, 2002.

[Hugh82] R. J. M. Hughes, "Super-combinators a new implementation method for applicative languages", in *Proceedings of the 1982 ACM symposium on LISP and functional programming*, LFP '82, pp. 1–10, New York, NY, USA, 1982, ACM.

[Hugh84] R. J. M. Hughes, *The Design and Implementation of Programming Languages*, Ph.D. thesis, Oxford University, September 1984.

[John84] Thomas Johnsson, "Efficient compilation of lazy evaluation", in *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, SIGPLAN '84, pp. 58–69, New York, NY, USA, 1984, ACM.

[John85] Thomas Johnsson, "Lambda lifting: transforming programs to recursive equations", in *Proc. of a conference on Functional programming languages and computer architecture*, pp. 190–203, New York, NY, USA, 1985, Springer-Verlag New York, Inc.

[Jone91] Simon L. Peyton Jones and David R. Lester, "A Modular Fully-lazy Lambda Lifter in HASKELL", *Softw., Pract. Exper.*, vol. 21, no. 5, pp. 479–506, 1991.

[Lieb83] Henry Lieberman and Carl Hewitt, "A real-time garbage collector based on the lifetimes of objects", *Commun. ACM*, vol. 26, no. 6, pp. 419–429, June 1983.

[LLVM] "Accurate Garbage Collection with LLVM", `http://llvm.org/docs/GarbageCollection.html`. Accessed: 2013 August 25.

[McCa60] John McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I", *Commun. ACM*, vol. 3, no. 4, pp. 184–195, April 1960.

[Mont70] Richard Montague, "Pragmatics and Intensional Logic", *Synthese*, vol. 22, no. 1-2, pp. 68–94, 1970.

[Mora08] Marco T. Morazán and Ulrik P. Schultz, "Optimal Lambda Lifting in Quadratic Time", in Olaf Chitil, Zoltán Horváth and Viktória Zsók, editors, *Implementation and Application of Functional Languages*, pp. 37–56, Springer-Verlag, Berlin, Heidelberg, 2008.

[Niel00] Lasse R. Nielsen, "A Denotational Investigation of Defunctionalization", 2000.

[Peyt87] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[Pott06] François Pottier and Nadji Gauthier, "Polymorphic typed defunctionalization and concretization", *Higher-Order and Symbolic Computation*, vol. 19, pp. 125–162, 2006.

[Reyn72] J. C. Reynolds, "Definitional interpreters for higher-order programming languages", in *Proceedings of* 25^{th} *ACM National Conference*, pp. 717–740, 1972.

[Rond94] P. Rondogiannis and W. W. Wadge, "Higher-order dataflow and its implementation on stock hardware", in *Proceedings of the 1994 ACM symposium on Applied computing*, SAC '94, pp. 431–435, New York, NY, USA, 1994, ACM.

[Rond97] P. Rondogiannis and W. W. Wadge, "First-order functional languages and intensional logic", *Journal of Functional Programming*, vol. 7, no. 1, pp. 73–101, 1997.

[Rond99] P. Rondogiannis and W. W. Wadge, "Higher-Order Functional Languages and Intensional Logic", *Journal of Functional Programming*, vol. 9, no. 5, pp. 527–564, 1999.

[Sche88] Bill Schelter and Michael Ballantyne, "Free AI development system: Kyoto Common LISP", *AI Expert*, vol. 3, no. 3, pp. 75–77, March 1988.

[Turn79] D. A. Turner, "A New Implementation Technique for Applicative Languages.", *Softw., Pract. Exper.*, vol. 9, no. 1, pp. 31–49, 1979.

[Unga84] David Ungar, "Generation Scavenging: A non-disruptive high performance storage reclamation algorithm", *SIGSOFT Softw. Eng. Notes*, vol. 9, no. 3, pp. 157–167, April 1984.

[Wads71] Christopher P. Wadsworth, *Semantics and Pragmatics of the Lambda-Calculus*, Ph.D. thesis, Oxford University, September 1971.

[Went90] E. P. Wentworth, “Pitfalls of conservation garbage collection”, *Softw. Pract. Exper.*, vol. 20, no. 7, pp. 719–727, July 1990.

[Wils92] Paul R. Wilson, “Uniprocessor Garbage Collection Techniques”, in *Proceedings of the International Workshop on Memory Management*, IWMM ’92, pp. 1–42, London, UK, UK, 1992, Springer-Verlag.

[Wils95] Paul R. Wilson, Mark S. Johnstone, Michael Neely and David Boles, “Dynamic Storage Allocation: A Survey and Critical Review”, in *Proceedings of the International Workshop on Memory Management*, IWMM ’95, pp. 1–116, London, UK, UK, 1995, Springer-Verlag.

[Yagh84] Ali A. Yaghi, *The Intensional Implementation Technique for Functional Languages*, Ph.D. thesis, University of Warwick, September 1984.