Annales Univ. Sci. Budapest., Sect. Comp. 36 (2012) 117–142

COMPILING HASKELL TO JAVASCRIPT THROUGH CLEAN'S CORE

László Domoszlai (Budapest, Hungary)
Rinus Plasmeijer (Nijmegen, The Netherlands)

Communicated by Zoltán Horváth

(Received December 15, 2011; accepted February 10, 2012)

Abstract. Crossing the borders of languages by letting them cooperate on source code level has enormous benefits as different languages have distinct language features and useful libraries to share. This is particularly true for the functional programming world, where languages are in constant development being the target of active research. There already exists a double-edged compiler frontend for the lazy functional languages Haskell and Clean, which enables the interoperation of features of both languages. This paper presents a program transformation technique to solve the same problem at another level by transforming STG, the core language of the flagship Haskell compiler GHC to SAPL, the core language of Clean. By this transformation (1) we have made a Haskell to JavaScript compiler with the advantage that Haskell applications can now be executed e.g. in a browser; (2) since there already existed a Clean to JavaScript compiler, under certain limitations, one can mix Clean and Haskell code because they are compiled to the same target code using the same run-time system and calling convention; (3) one can more easily compare the core code generated by the two compilers and measure their execution times.

Key words and phrases: Clean, Haskell, GHC, STG, SAPL, transformation.
2010 Mathematics Subject Classification: 68N18, 68N20.
1998 CR Categories and Descriptors: D1.1, D.3.3.
The research of the first author was supported by the European Union and the European Social Fund under the grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003.

1. Introduction

Equivalence questions of core languages of different lazy functional languages have not attracted much attention so far despite their many theoretical and practical use. First of all, an obvious theoretical question raises: does the core language matter? One conjectures that the short answer is no being all of them some kind of enriched lambda calculus, but even if it is true how does the core language affect the run-time properties of the generated code?

Furthermore, the core languages often serve as the input of compilation to alternative target platforms like Java or JavaScript. This fact promptly provides a natural practical use: by transforming a core language into another one which possesses compiler to an alternative target platform, the source language gets that platform as well. This idea is accomplished at a different level of abstraction in [8]. There was implemented a double-edged compiler frontend for Clean to be able to compile Haskell code as well. In that project, however, the primary goal was to provide interoperability of language features and libraries for both of the languages.

Why do we need a Haskell to SAPL compiler when there is a compiler which is able to generate SAPL from both Haskell and Clean code? Apparently, there is a theoretical usefulness of doing the same thing in a different way. Yet, the main points are the above-mentioned equivalence question and the limitation of this double-edged frontend: it currently compiles only Haskell98 code, which comprises only a limited feature set.

We chose STG, one of the core languages of the flagship Haskell compiler GHC, as source language of the transformation, because of the many interesting language features of GHC. Strictly speaking, the STG language is not the core language of the GHC compiler. There exists a *GHC Core*, which is a very small, explicitly-typed language. However, STG can be considered as a variant of it, because this core language is transformed to STG as the last step of a series of intermediate representations. Moreover, STG is the language of the so called **S**pineless **T**agless **G**-machine (STG) to which GHC compiles.

Our target language is SAPL, one of the core languages of Clean. This latter has the advantage of being the platform of a highly efficient interpreter technology [9] and JavaScript compiler [6].

However, these languages are interesting even in themselves. They are both typeless, but while SAPL can be considered as a high level lambda calculus designed for efficient interpretation, STG is designed to be compiled to machine code, and to make code generators simple. SAPL has a definite call-by-need evaluation strategy and many language features to enable interoperability with other languages, e.g. syntactic sugar for records to be able to perfectly com-

municate using JSON. In contrast, STG is at a much lower level of abstraction. It does not even have its own evaluation strategy, it is already compiled out by the GHC compiler frontend and the STG code is augmented with explicit force instructions.

In this paper we try to investigate the possibilities of translating STG to SAPL. Our primary goal is to develop a reliable transformation technique which enables the mixing of SAPL code of different sources under minimal restrictions. Clean profits from this property of the transformation. As for Haskell, let this transformation be the source language of one of the the most compelling and efficient JavaScript code generators. In Figure 1 an overview of the Clean and Haskell compilers is presented. It shows how the SAPL language fits the picture: both compiler frontends have been tapped to convert their cores (internal data structures) into SAPL. Later on this intermediate language, as a common multiply, is used to generate JavaScript code. Being this common SAPL the source language of the JavaScript compiler it has the enormous advantage that the same run-time system and calling convention is used for both languages, thereby the foundations of interoperability have been made.

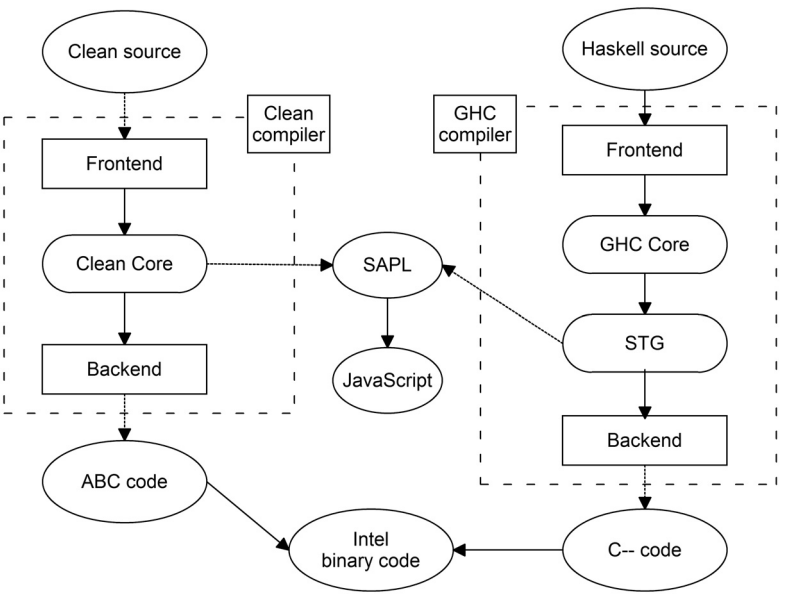

Figure 1. An overview of the Clean and GHC compilers augmented with the SAPL/JavaScript backend

2. Introduction into SAPL

SAPL stands for **S**imple **A**pplication **P**rogramming **L**anguage. This is the target of the transformation, a core lazy purely-functional language. Before we explain the language in details, let us present some simple examples to grab the main idea. Consider the Clean code of the following factorial and summation functions:

```
fact 0 = 1
fact n = n * fact (n-1)

:: List a = Nil | Cons a (List a)

sum Nil = 0
sum (Cons x xs) = x + (sum xs)
```

The equivalent function definitions in SAPL are the following:

```
fact n = if (eq n 0) 1 (mult n (fact (sub n 1)))

:: List = Nil | Cons x xs
sum xxs = select xxs 0 (λx xs = x + sum xs)
```

Finally, the following `mappair` function written in Clean is a more complex example, which is also based on the use of pattern matching:

```
:: List a = Nil | Cons a (List a)
mappair f Nil         zs          = Nil
mappair f (Cons x xs) Nil         = Nil
mappair f (Cons x xs) (Cons y ys) = Cons (f x y) (mappair f xs ys)
```

This definition is transformed to the following SAPL function:

```
:: List = Nil | Cons x xs
mappair f as zs
  = select as Nil (λx xs =
          select zs Nil (λy ys = Cons (f x y) (mappair f xs ys)))
```

Basically, SAPL is obtained from Clean by removing type information (this is done after the type checking phase of the compilation, SAPL programs generated from Clean are type correct) and syntactic sugar. This includes the contraction of partial functions by the means of SAPL pattern matching contracts `if` and `select`, as it can be seen in these examples.

SAPL was originally constructed as a language of its own implementing language features to enable efficient interpretation [9]. Later a Clean like type definition style and other language features were adopted for readability and to allow for the generation of highly efficient JavaScript code [6]. Currently SAPL

source code can be generated using the Clean compiler, thus, considering the intermediate nature of SAPL, it can be regarded as a core language of Clean.

The formal definition of the language is given in [6], therefore we present here its main characteristics only ensuring thereby the clarity of the following sections. The simplified syntax of SAPL is illustrated in Figure 2.

D	::=	$:: x = C\ \vec{x} \mid \ldots \mid C\ \vec{x}$	(Algebraic Type Definition)
f	::=	$x\ \vec{x} = e$	(Function Definition)
e	::=	$x\ \vec{e}$	(Application)
	$\mid$	**select** $e\ \vec{e}$	(Select Expression)
	$\mid$	**if** $e\ e\ e$	(If Expression)
	$\mid$	**let** $\vec{b}$ **in** e	(Let Expression)
	$\mid$	$\lambda\ \vec{x} = e$	(Lambda Expression)
	$\mid$	x	(Variable)
	$\mid$	l	(Literal)
b	::=	$x = e$	(Let Binding)

Figure 2. Core syntax of SAPL

Select Expressions `Select` expressions are intended to perform pattern matching. The `select` keyword is used to make a case analysis on the data type of its first argument, thus to accomplish this, the first argument is reduced to *head normal form* before. The remaining arguments handle the different constructor cases in the same order as they occur in the type definition (all cases must be handled separately). Each case is a function that is applied to the arguments of the corresponding constructor.

Lambda Expressions Only the arguments of a `select` expression can contain lambda expressions in SAPL. All other nested lambda expressions must be lifted to the top-level.

If Expressions An `if` expression can be considered as a normal function which has strict semantics in its first argument, in its *predicate*. It returns its second or third argument depending on the predicate which must be reducible to a Boolean value.

Let Expressions Only constant (non-function) `let` expressions are allowed that may be mutually recursive (for creating cyclic expressions).

Applications SAPL, in contrast to STG, does not require applications to be saturated, and arguments of applications are allowed to be other applications (compound applications).

Function Definitions Function definitions can be split into two kinds. SAPL distinguishes normal functions and *constant applicative forms* (CAF). CAFs cannot have any arguments and the ':=' notation is used for them.

Algebraic Type Definitions Explicitly providing algebraic type definitions is essential being the constructor indexes used by the `select` construct. In addition, the arity of data constructors carries important information for the code generator.

3. Introduction into the STG language

The **S**hared **T**erm **G**raph (STG) language, one of the core languages of GHC, the language of the **S**pineless **T**agless **G**-machine, is the source of the transformation. To illustrate its main idea, let us consider the source code of the summation function of the previous section (to gain this piece of code, one has to provide explicit type definition for the `sum` function in Haskell, otherwise it generates a less concise, generalized form of the function by the means of type classes). The STG counterpart of the factorial example is given in Figure 3.

```
Main.fact =                                                                    1
  λr srt:(0,*bitmap*) [ds_sCK]                                                 2
    case ds_sCK of wild_sCP {                                                  3
      GHC.Types.I# ds1_sCN →                                                   4
        case ds1_sCN of ds2_sDb {                                              5
          __DEFAULT →                                                          6
            let {                                                              7
              sat_sDc =                                                        8
                λu srt:(0,*bitmap*) []                                         9
                  let {                                                        10
                    sat_sDd =                                                  11
                      λu srt:(0,*bitmap*) []                                   12
                        let { sat_sDe = NO_CCS GHC.Types.I#! [1];              13
                        } in GHC.Num.- GHC.Num.fNumInt wild_sCP sat_sDe;       14
                  } in Main.fact sat_sDd;                                      15
            } in GHC.Num.* GHC.Num.fNumInt wild_sCP sat_sDc;                   16
          0 → GHC.Types.I# [1];                                                17
        };                                                                     18
    };                                                                         19
```

Figure 3. The STG code of the factorial function generated by GHC version 7.2.1 without any optimization turned on

```
Main.sum =
    λr srt:(0,*bitmap*) [ds_sL7]
        case ds_sL7 of wild_sMv {
          Main.Nil → GHC.Types.I# [0];
          Main.Cons x_sLb xs_sLc →
              let { sat_sMu = λu srt:(1,*bitmap*) [] Main.sum xs_sLc;
              } in  GHC.Num.+ GHC.Num.fNumInt x_sLb sat_sMu;
        };
```

Conceptually, the STG language is an enriched lambda calculus, furthermore, it can be regarded as a variant of *administrative normal form* (ANF) [7] as (1) it allows only constants and variables to serve as arguments of function applications (flat applications), and (2) it requires the result of a non-trivial expression to be assigned to a `let`-bound variable or returned from a function.

Being the full language description rather extensive, a simplified syntax is presented only in Figure 4, which is sufficient for our purposes. In the following we highlight the salient characteristics of the language; the full syntax along with well-defined *operational* semantics are given in [14].

f	::=	$x = \mathrm{e}$	(Function Definition)
e	::=	$x\ \vec{a}$	(Application)
	|	$C\ \vec{a}$	(Constructor Application)
	|	**case** e **of** $x\ c$	(Case Expression)
	|	**let** $\vec{b}$ **in** e	(Let Expression)
	|	$\lambda\pi\ \vec{x} = e$	(Lambda Expression)
a	::=	x	(Variable)
	|	l	(Literal)
b	::=	$x = e$	(Let Binding)
c	::=	$\vec{m} \mid \vec{n}$	(Case Alternative)
m	::=	$C\ \vec{x}$ –> e	(Algebraic)
n	::=	l –> e	(Primitive)

Figure 4. Core syntax of STG

Case Expressions `Case` expressions are used to perform pattern matching. In contrast to SAPL it handles both algebraic and primitive data types and this is the only construct where data value is ever forced in STG. This expression has a rather unique form, a bit of extra syntax attached to it,

a variable to the right of the `of` keyword which indicates the evaluated value of the pattern matched expression. `Case` alternatives may contain a default branch as it can be seen in Figure 3, line 6. This *default case* is always the first, if it is there at all.

Let Expressions `Let` expressions begin with one of the `let` or the STG specific `let-no-escape` keywords. This latter is used when the compiler guarantees that the bounded variable can be used without a full closure creation for the expression it is bound to. `Let` expressions can be mutually recursive and always bind variables to constructs resulting in a *thunk* (suspended computation): lambda abstractions, constructor applications or other `let` expressions.

Lambda expressions Lambda expressions contain special hints for the code generator, the so called update flag. This flag is the first character (denoted by π in Figure 4) after the backslash, which indicates a lambda expression. The arguments of lambda abstractions are given in square brackets; if there are no arguments, that lambda expression simply denotes a thunk. For further information about the connection of lambdas, thunks and update flags please refer to [14].

Applications There are two flavors of applications in STG. Normal functions take their arguments simply enumerated after the name of the function, while constructors and primitive operators take their arguments in square brackets. This is not just a stylistic change: STG requires that all constructors and primitive operators are *saturated* (an application is saturated if it gives the function exactly the number of arguments it expects); this is indicated by the square brackets.

Function Definitions Function definitions are named lambda expressions in STG, that is the `f` $= \lambda\pi$ `a1 a2 ... an` $\rightarrow$ `body` form is used instead of the usual `f a1 a2 ... an` $=$ `body` style.

4. Salient differences between STG and SAPL

In this section we would like to emphasize two significant, non-syntactic differences of the languages. The first one is the presence and absence of definitions of algebraic data types and, what is related to this issue, of saturated applications. The presence of data constructors provides two pieces of information: (1) the constructor indexes and (2) the arity of the constructors. The arity of constructors carries information for the code generator, and it helps

with deciding on whether a given constructor curried or not. STG solves this problem by requiring constructor applications to be saturated. As for the constructor indexes, this information is needed by SAPL `select` expressions: in contrast to STG, where in a `case` expression constructors must be given by name, `select` is parametrized with the help of constructor indexes. This results in simpler language element for more information in turn. However, the extra information is not a waste: e.g. it enables the JavaScript code generator to accelerate constructor applications.

The second difference is more essential: SAPL pattern matching constructs (`if` and `select` expressions) and primitive functions have strict semantics, that is it realizes a definite *call-by-need* evaluation strategy, while STG does not even have its own evaluation strategy, it is already compiled out by the GHC compiler frontend and the STG code is augmented with explicit force instructions by the means of `case` expressions. As a consequence, although we know that Haskell also uses call-by-need evaluation strategy, there may have subtle differences in the actual evaluation order of the languages. In a pure, effect-free setting, they must produce the same result, however with the presence of non-pure functions, it may result in different output.

This remark has of primary importance considering that the following rewrite rules change the evaluation order of the original STG program, thus we must declare that the transformation is valid only in an effect-free environment. To determine the impact of the transformation on non-pure functions further investigation is needed, which is not part of the current paper.

5. Transforming STG to SAPL

In this section we present a series of program transformations which, by applying them on an STG program, result in SAPL code. We use the following simple *factorial function* to illustrate certain steps:

```
fact :: Int → Int
fact 0 = 1
fact n = n * fact (n - 1)
```

Before digging into the explanation of the rewrite rules, let us have a look at Figure 3, which contains the STG code of the factorial function. The code is intuitively straightforward considering the brief syntax description of the previous section, however it contains some oddness that still needs explanation.

- Lambda expressions now contains, in addition to the update flag and the argument list, a new element, which begins with `srt:()`. This is the so

called *static reference table* required for the garbage collection of static elements, e.g. CAFs.

- STG code may contain various annotations related to profiling. Since we were not compiling with profiling, the only annotation we can see here is `NO_CSS`, which means that no *cost-centre stack* attached to the given thunk.
- The exclamation mark given as the last character of the function part of an application indicates constructor application.

These annotations carry information for the code generator only, we do not need them for our transformation, thus, as a preliminary step, they are removed from the token stream. We also remove the update flags of the lambda expressions and then, finally, `let-no-escape` keywords substitute for `let` keywords (it is a valid transformation, `let-no-escape` expressions are simple `let` expressions with additional information for the code generator).

After this step the *abstract syntax tree* (AST) of the SGT program is produced and the following rewrite rules are applied step by step in the given order. This technique is sometimes called *compilation by transformation* [15], although because the intermediate code is neither SAPL nor STG, strictly speaking, the certain transformation steps are not *correctness-preserving*.

Transformation of named lambda expressions

Function definitions are named lambda expressions in STG, which can be easily turned into SAPL function definitions by the following rewrite rule:

$$f = \lambda\ [x_1 \ldots x_n] = b \rightarrow f\ x_1 \ldots x_n = b$$

This rule becomes slightly different in the case of empty argument list, because according to the semantics of Haskell, top-level functions with an empty argument list are just CAFs:

$$f = \lambda\ [] = b \rightarrow f := b$$

Extraction of algebraic type definitions

In SAPL, algebraic data type definitions must be explicitly provided. In contrast, STG code does not contain explicit type definitions, but a pre-generated function for every data constructor. Fortunately, these functions have a special form, which enables us to recognize and turn them into constructor definitions. Consider the following type definition in Haskell:

data `T = A Int Int | B`

After generating STG and applying the previous steps of transformation we gain this intermediate code:

```
Main.A eta_B2 eta_B1 = Main.A [eta_B2 eta_B1];
Main.B = Main.B [];
```

Considering this example, the rewrite rule is straightforward now:

$$f\ x_1 \ldots x_n = f\ [x_1 \ldots x_n] \rightarrow :: \text{generated} = f\ x_1 \ldots x_n$$

These functions are converted into a type called `generated`, which has only one data constructor. Later in this step these data constructors are merged and the original functions are removed. In our example, the final result of this step is as follows:

```
:: generated = Main.A eta_B2 eta_B1 | Main.B;
```

The type of the data constructors cannot be recognized this way (therefore data constructors of different types will be collected into the `generated` type), however it is not even necessary. In SAPL, type definitions are used to (1) determine the constructor index of a given constructor and to (2) determine the arity of a given constructor. The second requirement is obviously satisfied, so we have to consider only the first one. The first one is required to ensure that the data constructors occur in a `select` statement and have different constructor indexes to avoid collision. This criterion is met and even more: *every* data constructor will have different constructor indexes.

However, being the transformation valid is merely part of the picture. `Select` expressions, which will handle algebraic pattern matching in SAPL eventually, must have as many arguments as the number of data constructors of the type of the expression being the subject of pattern matching. When many data constructors are defined in a given program module, because they are treated as the constructors of one common type, `select` expressions will be cluttered with so many (unnecessary) arguments.

A possible solution is to discover groups of constructors by defining an equivalence relation. We can say that two constructors are in the some group (have the same type) if they are both alternatives of the same `case` expression. Computing the transitive closure of the so gained groups we eventually get a valid grouping of data constructors. The current prototype implementation of the compiler does not implement this algorithm.

Saturation information removal

SAPL does not differentiate between applications, because it does not require any of them to be saturated as STG does. In this small step (after we used

this information in the previous one) the saturation information of applications is removed:

$$f\ [x_1 \ldots x_n] \to f\ x_1 \ldots x_n$$

Elimination of explicit thunk creation

STG provides a way to explicitly denote thunks using lambda expressions with an empty argument list. In SAPL, thunks are recognized automatically during code generation/interpretation, furthermore lambda expressions without arguments are not allowed by the language. This step removes them (line 9 and 12 in our example).

$$\lambda\ [] = b \to b$$

Elimination of the computation logic of `case` expressions

In STG, `case` expressions encode both pattern matching and computation. It evaluates a given expression, binds a variable to the evaluated value and matches the value to the `case` alternatives. However, considering that a `let` binding creates a thunk, moving the pattern matched expression into an outer binding would cause the same effect. In this way we can get rid of this rather unique `case of` construct by substituting it for a normal `case` expression embedded into a `let` expression. The rewrite rule is the following:

$$\texttt{case}\ e\ \texttt{of}\ x\ \vec{b} \to \texttt{let}\ x = e\ \texttt{in}\ \texttt{case}\ x\ \vec{b}$$

The intermediate code after this step is given in Figure 5. It is still cluttered by numerous `let` expressions, but the picture is getting clearer.

Elimination of variable-to-variable bindings

After the previous step we introduced several new `let` bindings. However, if we take a careful look at Figure 5, it can be seen that we introduced variable to variable bindings only. Actually, the expression in a `case of` statement can be an application or even another `case` expression as well, but in the most frequently occurring case it is only a variable. When this is so, we can further simplify the AST by substituting `let`-bound variables in the nested expression. This transformation is sometimes called *copy propagation*. Please notice that this rule would remain valid even if x_2 would be an arbitrary expression for the price of losing call-by-need semantics.

$$\texttt{let}\ b_1, \ldots, x_1 = x_2, \ldots, b_n\ \texttt{in}\ e \to \texttt{let}\ b_1, \ldots, b_n\ \texttt{in}\ e[x_1 := x_2]$$

```
Main.fact ds_sCK =                                                          1
  let {                                                                     2
    wild_sCP = ds_sCK                                                       3
  } in case wild_sCP {                                                      4
      GHC.Types.I# ds1_sCN →                                                5
        let {                                                               6
          ds2_sDb = ds1_sCN                                                 7
        } in case ds2_sDb {                                                 8
               __DEFAULT →                                                  9
                  let {                                                     10
                    sat_sDc =                                               11
                      let {                                                 12
                        sat_sDd =                                           13
                          let { sat_sDe = GHC.Types.I# 1;                   14
                          } in GHC.Num.- GHC.Num.fNumInt wild_sCP sat_sDe;  15
                      } in Main.fact sat_sDd;                               16
                  } in GHC.Num.* GHC.Num.fNumInt wild_sCP sat_sDc;          17
             0 → GHC.Types.I# 1;                                            18
           };                                                               19
      };                                                                    20
```

Figure 5. The intermediate code after five steps

In our example, this transformation removes line 3 and 7 and changes line 4, 8, 15 and 17. The `let` statements with empty binding list, caused by this transformation, will be removed in a subsequent step.

Elimination of forcing-only `case` expressions

Since only `case` expressions force evaluation in STG, it sometimes happens that its pattern matching facility is not even used. When this is so, only a *default case* exists. In SAPL there is no language construct to express default case at pattern matching; these expressions must be removed from the AST:

$$\texttt{case}\ x\ \texttt{\_\_DEFAULT} \texttt{->}\ e \rightarrow e$$

By this transformation the evaluation of the given expression is delayed, but, as long as we regard pure functions only, it is certainly safe by the considerations of Section 4.

Lambda lifting

SAPL allows lambda expressions as the arguments of a `select` statement only. STG, however, encodes local functions as lambda expressions occurring

in `let` bindings. In this step these `let` bindings are lifted to the top-level as new function definitions:

$$\begin{aligned}\texttt{let}\ b_1, \ldots, f = \lambda[x_1 \ldots x_n] = e_1, \ldots, b_n\ \texttt{in}\ e_2 \\ \rightarrow \texttt{let}\ b_1, \ldots, b_n\ \texttt{in}\ e_2[f := f' FV_1 \ldots FV_m]\ ,\end{aligned}$$

where f' is a new, uniquely generated function name, FV_i is the ith local free variable (which is not free in the original function) of the expression assigned to f, and m denotes the number of such free variables. The definition of the f' function is as follows:

$$f'\ FV_1\ \ldots FV_m\ x_1 \ldots x_n = e_1$$

Application inlining

The STG machine requires function and constructor arguments to be atoms (variables or constants). This constraint implies that all sub-expressions are explicitly named and the evaluation order is explicit. This is well suited to the pursuit of making the code generator as simple as possible. However, some research indicates that if the bodies of functions or `let` bindings are mostly small (which is the consequence of flat applications), the interpretation overhead is relatively large [9].

In this step we invert the decomposition of non-flat applications, which is done during the translation of GHC Core into STG. That time new `let` bindings were added for the non-trivial arguments of applications; now `let`-bound variables are inlined. We have to be careful, however. To preserve the sharing property of thunks, only those `let` bindings can be inlined which occur (1) only once and (2) only as the argument of an application. Furthermore, a `let` can be inlined only if it is not mutually recursive. When these conditions are satisfied, the rewrite rule is the following:

$$\texttt{let}\ b_1, \ldots, x = e', \ldots, b_n\ \texttt{in}\ e \rightarrow \texttt{let}\ b_1, \ldots, b_n\ \texttt{in}\ e[x := e']$$

Subsequently, `let` expressions with an empty binding list must be removed and the transformation step must be repeated until there is no `let` binding which satisfies the necessary conditions. After three iterations of this rule, our example looks rather concise (the long line of embedded applications is broken into multiple lines) as it can be seen in Figure 6.

Lifting of compound expressions of `let` bindings

SAPL does not allow `let` expressions to nest other `let` expressions, neither in bindings nor in spine. In the previous step we eliminated some of them by

```
Main.fact ds_sCK =                                                          1
  case ds_sCK {                                                             2
    GHC.Types.I# ds1_sCN →                                                  3
      case ds1_sCN {                                                        4
        __DEFAULT →                                                         5
          GHC.Num.* GHC.Num.fNumInt ds_sCK                                  6
                (Main.fact                                                  7
                    (GHC.Num.- GHC.Num.fNumInt ds_sCK (GHC.Types.I# 1)));   8
        0 → GHC.Types.I# 1;                                                 9
      };                                                                    10
  };                                                                        11
```

Figure 6. The intermediate code after inlining applications

inlining `let` bindings when they were applications. The remaining of such `let` expressions, unfortunately, cannot be managed so easily; some bindings must be lifted as a top-level function. These are the cases when a binding contains another `let`, or a `case` expression. The following rewrite rule uses the notations introduced at lambda lifting:

$$\texttt{let } b_1, \ldots, f = e_1, \ldots, b_n \texttt{ in } e_2 \\ \rightarrow \texttt{let } b_1, \ldots, b_n \texttt{ in } e_2[f := f' FV_1 \ldots FV_m] \ ,$$

where e_1 is a `case` or `let` expression and the definition of f' is as follows:

$$f' \ FV_1 \ \ldots FV_m = e_1$$

After the lifting of such expressions, the previous step must be repeated to inline the newly introduced applications.

Fusion of nested `let` expressions

So far, we eliminated `let` expressions occurring in `let` bindings and inlined most of the nested ones. However, to preserve call-by-need semantics, bindings are allowed to be inlined when they occur only once in the nested body, therefore the intermediate code still can contain nested `let` expressions. To avoid them, the bindings of the nested expression must be lifted and merged with the bindings of its container. In this case, mutually recursive `let` bindings can be moved safely, because they will not be broken up.

$$\texttt{let } b_1, \ldots, b_n \texttt{ in let } d_1, \ldots, d_m \texttt{ in } e \\ \rightarrow \texttt{let } b_1, \ldots, b_n, \ d_1, \ldots, d_m \texttt{ in } e$$

This transformation finally results in an intermediate code, which does not contain nested `let` expressions.

Transformation of pattern matching logic

Pattern matching is performed by `case` expressions in STG. So far we eliminated the computation logic of this construct along with its special utilization to force evaluation. This time the two flavors of `case` expressions, pattern matching of algebraic and primitive types, are separated and translated into SAPL `select` and `if` expressions, respectively. Being both corresponding SAPL constructs strict in their first argument, that is force evaluation, this transformation preserves the semantics of the original program. The type of a given `case` expression can be clearly recognized by the analysis of the left hand side of the `case` alternatives: if any of them is literal, we have to convert it to an `if` expression, otherwise we deal with the algebraic type.

- Pattern matching on primitive types is converted to SAPL `if` expressions. These expressions have three arguments in this order: (1) predicate (2) true-branch and (3) false- or else-branch. The algorithm is the following: the `case` alternatives are converted to nested `if` expressions, so we have to determine first the else-branch of the inmost `if` expression. It will be the default `case` alternative or the last `case` alternative if a default case does not exist. Then the remaining alternatives, from top to bottom, are converted to `if` expressions: every such expression will be the else-branch of the previous one. The predicates are translated into an application of the primitive `eq` function.

```
case x
  l1    -> e1      →  if (eq x l1) e1
  l2    -> e2            (if (eq x l2) e2
        ⋮                        ⋮
  ln-1  -> en-1                (if (eq x ln-1) en-1 en)...)
  ln    -> en
```

- Algebraic data constructors are already discovered by the previous steps, thus we can easily determine the constructor indexes. This information is vital as it defines the order of the arguments of the `select` statement which is generated from such a `case` expression. We also need to know the arity of the data constructors; fortunately, this information is both available in the definition of data constructors and in the left hand side of the `case` alternatives.

  The GHC compiler does not generate a `case` alternative for every data constructor of a given type. The `case` expression generated from a partial function will contain a default case alternative for the non-defined constructors. In addition to this, we merged constructors of different types,

thus the positions of the data constructors belonging to other types must be filled in the `select` statement. We will use the primitive `nomatch` function for this purpose. With this remark, these functions are neglected from the rewrite rule for the sake of readability.

$$\begin{array}{lcl} \texttt{case}\ x & \rightarrow & \texttt{select}\ x \\ \quad C_1\ \vec{x_1}\ \texttt{->}\ e_1 & & \quad \lambda\ x_{\vec{I(1)}}\ =\ e_{I(1)} \\ \quad\quad \vdots & & \quad\quad \vdots \\ \quad C_n\ \vec{x_n}\ \texttt{->}\ e_n & & \quad \lambda\ x_{\vec{I(n)}}\ =\ e_{I(n)} \end{array}$$

The $I(n)$ function yields the index of the constructor in the list of `case` alternatives by the global constructor index n.

```
Main.fact ds_sFz = select ds_sFz (λ ds1_sFC =                                    1
    if (eq ds1_sFC 0) (GHC.Types.I# 1)                                           2
       (GHC.Num.* GHC.Num.fNumInt ds_sFz                                         3
          (Main.fact (GHC.Num.- GHC.Num.fNumInt ds_sFz (GHC.Types.I# 1)))))      4
```

Figure 7. The final result of the transformation

This final step results in a valid SAPL function (Figure 7). In the following section we discuss interoperability issues, that is the differences between SAPL functions generated from Clean and GHC through STG.

6. Interoperability issues between Haskell and Clean

There is another reason choosing the factorial function as primary example of this paper behind its simplicity: it can be compiled by both Haskell and Clean compilers without any modification. This property gives us the opportunity to compare objectively the generated SAPL functions.

In [8], van Groningen et al, identify the most salient differences between Clean and Haskell. These are: modules, functions, macros, newtypes, type classes, uniqueness typing, monads, records, arrays, dynamic typing, and generic functions. Studying this list carefully one can recognize that after the elimination of types and syntactic sugar we have to deal with the different low level representations of basic constructs only, e.g. the representations of basic types, arrays, records, tuples and so on.

First of all, it must be defined what we mean by the term interoperability here. Consider the following setting: we have two pieces of programs, one in Haskell and one in Clean. In both codes we define the same type (it is not always possible, see [8], but we will study interoperability only for those cases when compatible types exist). We have different functions working on this type written in Clean and Haskell. What happens if we composite them after their translation to SAPL? This is the primary question this section wants to answer.

The following SAPL code snippet shows the code of the factorial function produced by the Clean compiler:

```
Main.fact x_0 = if (eq x_0 0) 1 (mult x_0 (Main.fact (sub x_0 1)))
```

Comparing it to Figure 7, although the two codes seem completely different, we can identify only two non-isomorph differences and only one of them is essential. The less significant difference is the usage of type classes in GHC. The `GHC.Num.-` and `GHC.Num.*` functions are responsible for subtracting and multiplying numbers. They can work on any kind of number; their first argument is a type class instance, a dictionary (`GHC.Num.fNumInt` in this example, the `Int` instance of the `Num` type class). Substituting them for the wrapper functions `add'` and `mult'`, helps us to clear the picture:

```
Main.fact ds_sFz = select ds_sFz (λ ds1_sFC =
    if (eq ds1_sFC 0) (GHC.Types.I# 1)
                      (mult' ds_sFz (Main.fact (add' ds_sFz (GHC.Types.I# 1)))))
```

Primitive types

Now we can see the fundamental difference: Haskell primitive types are *boxed*, they are wrapped by a data constructor (`GHC.Types.I#` for the type `Int`). The `select` expression in this example unboxes the wrapped value. These boxing semantics are important properties of Haskell. For more on this, please refer to [14].

These semantics obviously cause interoperability problems. If we want to call this factorial function generated from STG (SAPL* in the following) from SAPL, we have to box the argument and unbox the return value:

```
Clean.fact x_0 = select (Haskell.fact (GHC.Types.I# x_0))) (λ r = r)
```

What could we do here? If we insist on the primitive types being fully compatible at this level, we can devise rewrite rules to unite representations. By removing `case` expressions responsible for unboxing

$$\texttt{case}\ x\ \texttt{GHC.Types.I\#}\ l\ \texttt{->}\ e\ \rightarrow\ e[p'\ :=\ p]$$

and by replacing applications, responsible for boxing, with their primitive value

$$\texttt{GHC.Types.I\#}\ l \to l$$

we gain a sound transformation for boxed integers (p' denotes a primitive operator which works on boxed type, and p is its unboxed counterpart).

However, what we win on the compatibility we lose on performance. If boxing semantics are removed, optimizations of the GHC compiler cannot be used any more as boxing of primitive types plays an important role in the handling of strictness (see Section 7).

Algebraic data types

In theory, instances of ADTs could be passed between SAPL and SAPL*, the only restriction that constructor indexes must match. Unfortunately, this condition is currently not satisfied since transformation from STG to SAPL* merges the data constructors of different types. The implementation of the grouping algorithm given in the previous section, however, would solve this issue sufficiently.

Records

SAPL has special syntax to define records. Clean records are converted to SAPL records during compilation. SAPL records, however, are just syntactic sugar to allow distinguishing between constructors and records [6], they are simple ADTs. GHC also represents records as ADTs, that is, with the remarks of the preceding paragraph, records are compatible at this representation level.

Lists and tuples

Lists and tuples are represented as ADTs in both SAPL and SAPL*. The only difference is in constructor names. SAPL* uses rather special names to refer to these types. For *Nil* and *Cons* the keywords `[]` and `:` are used, tuples with different arities are denoted by `(,)`, `(,,)`, ... In contrast, SAPL uses the following compatible definitions:

```
:: predefined_List = predefined_Cons a1 a2 | predefined_Nil

:: predefined_Tuple = predefined_Tuple1 a1
:: predefined_Tuple = predefined_Tuple2 a1 a2
...
```

As for interoperability, only constructor indexes matter and this requirement is satisfied by these types.

Strings

The double-quoted form of a string literal in Haskell is just syntactic sugar for list notation. This is a fundamental difference to Clean, where strings are represented as unboxed arrays, that is simple objects. The difference is so essential that it can be solved only by runtime conversion. The two different string representations must be converted to each other by applying special conversion functions to them.

Arrays

Clean has extensive language support, including syntactic sugar, for the efficient handling of arrays. However, the syntactic sugar is completely eliminated by the Clean compiler frontend, and it appears as a set of primitive functions at the level of SAPL. Haskell has no built-in support, it provides arrays via a standard module. Because there is no special elements for arrays in none of the languages, the problem is simplified to using the same implementation through different APIs. In this case, a special SAPL implementation of Haskell `Data.Array` module must be provided, which ensures compatibility by using SAPL primitive functions for array creation and access.

Type classes

GHC generates ADTs, while Clean generates records from type classes. By the above-mentioned remarks, these are compatible at this representation level.

7. Benchmarks

So far we concentrated on a mere valid transformation technique. In this section we discuss performance issues and present comparison of run-times of JavaScript code generated from SAPL code gained from Clean and GHC through STG using the above described technique.

In Figure 8 the optimized STG code (using GHC -O2) of the factorial example is presented. Comparing it to Figure 3 one can identify two essential differences. First of all, the factorial function is split into two ones: separate versions are generated for strict and non-strict calls of the function. The entry function (`Main.fact`) is now responsible only for unboxing the argument and boxing the result. The actual computation logic is moved into a new function, which works on an unboxed primitive integer. As for the second difference, the

usage of type classes is replaced by direct applications of primitive functions (`-#, *#`). These modifications result in a completely different SAPL code:

```
Main.fact w_s1oZ = select w_s1oZ (λww_s1p2 = (GHC.Types.I# (Main.wfact ww_s1p2)))
Main.wfact ww_s1oS = if (eq ww_s1oS 0) 1 (*# ww_s1oS (Main.wfact (-# ww_s1oS 1)))
```

The new `Main.wfact` function, apart from the names of variables and primitive functions, is equivalent with the SAPL function generated by the Clean compiler. This optimization has a big impact on execution time and memory consumption, and actually this is how strictness is handled in Haskell [14].

```
Main.wfact =                                                          1
    λr [ww_s1oS]                                                      2
        case ww_s1oS of ds_s1oU {                                     3
          __DEFAULT →                                                 4
              case -# [ds_s1oU 1] of sat_s1t0 {                       5
                __DEFAULT →                                           6
                    case Main.wfact sat_s1t0 of ww1_s1oX {            7
                      __DEFAULT → *# [ds_s1oU ww1_s1oX];              8
                    };                                                9
              };                                                      10
          0 → 1;                                                      11
        };                                                            12
                                                                      13
Main.fact =                                                           14
    λr [w_s1oZ]                                                       15
        case w_s1oZ of w1_s1tP {                                      16
          GHC.Types.I# ww_s1p2 →                                      17
              case Main.wfact ww_s1p2 of ww1_s1p4 {                   18
                __DEFAULT → GHC.Types.I# [ww1_s1p4];                  19
              };                                                      20
        };                                                            21
```

Figure 8. The optimized STG code of the factorial example

Table 1 presents the results of an extensive range of benchmark programs. It shows the run-times of the binary programs, the run-times of their JavaScript counterpart, and finally, the memory usage and stack utilization of the binary programs. Every such test is evaluated using version 2.3 of the Clean compiler, GHC version 7.2.1 and the same GHC compiler version with the O2 flag. The JavaScript code is generated from SAPL, which is produced by the Clean compiler and by compiling the STG output of the GHC compiler using the above presented transformation.

All of the contemporary browsers have call stack size limitations, called *recursion limit*. As for Chrome, it is currently a bit more than 20.000. Functional

	JavaScript run-time in sec.			Binary run-time in sec.			Memory allocation in MB			Stack utilization in KB		
Program	**Cln**	**G**	**G_2**	**Cln**	**G**	**G_2**	**Cln**	**G**	**G_2**	**Cln**	**G**	**G_2**
Adjoxo	3.99	22.22	8.25	0.11	0.39	0.28	128	54	54	<10	<1	<1
Braun	**18.47**	15.82	16.36	0.47	0.82	0.68	383	605	549	**58**	<1	<1
Cichelli	1.83	21.12	4.49	0.06	0.30	0.13	27	59	34	<10	<1	<1
Clausify	9.93	26.61	14.47	0.28	0.62	0.39	28	385	156	<10	<1	<1
CountDown	3.16	18.50	3.70	0.06	0.31	0.09	53	102	45	<10	<1	<1
Fib	22.90	-	81.16	0.43	*5.78*	0.55	0	*1739*	0	<10	<1	<1
KnuthBendix	2.22	10.14	3.56	0.12	0.16	0.10	40	58	35	<10	<1	<1
Mate	64.91	*368.33*	109.90	1.80	*5.62*	2.12	*1070*	*1078*	251	<10	<1	3
MSS	11.88	88.23	13.35	0.10	1.05	0.21	18	521	26	<10	<1	<1
OrdList	**39.41**	24.94	23.84	2.06	1.09	0.94	573	816	739	>**1000**	<1	<1
PermSort	-	-	51.90	0.81	1.81	1.58	*990*	*1361*	1221	<20	<1	<1
Queens	31.42	140.82[1]	22.17	0.51	1.98	0.27	510	489	99	<25	**67**	3
$Queens_2$	24.52	**51.59**	26.46	0.57	1.23	1.03	481	676	593	<25	**67**	<1
SumPuz	52.87	*172.00*	50.82	1.01	*3.54*	2.28	840	*1305*	1046	<10	<1	<1
Taut	11.81	27.12	14.91	0.11	0.30	0.26	96	167	156	<10	<1	<1
While	6.56	23.69	10.08	0.17	0.39	0.27	201	179	136	<10	<1	<1

Table 1. Run-times of GHC and Clean compiled code running on an Intel Core 2 Duo T5870 PC clocking at 2GHz with 3 GB of memory. The JavaScript benchmarks, compiled from SAPL, were executed in Google Chrome 15.0.874.120. *Cln*, *G* and G_2 stand for Clean, GHC and GHC -O2, respectively

[1] because of high stack utilization Chrome must be run with the –js-flags=”–stack-size 2048” switch

programs inherently tend to use a huge amount of stack, so this limitation affects several benchmarks. With the exception of Queens benchmark, when we could overcome the problem by increasing moderately the stack size of Chrome, these ones must have been compiled by optimizing stack use instead of speed. In the table, this special compilation technique, along with the corresponding stack sizes, is indicated by bold faced numbers. Moreover, some JavaScript benchmarks fail because of high memory consumption. In these cases there are no results at the JavaScript run-times and italic font face is used to highlight the memory consumption numbers of the related binary programs. Please note that the stack/memory consumption of the JavaScript programs was not measured explicitly. The related problems were indicated by Chrome by the means of error messages.

The employed benchmark programs are part of [11], for details of the programs, including source code, see [12]. The pre-compiled SAPL, STG and JavaScript code can be obtained from [5].

Analyzing the results of Table 1 we can make some general observations about them:

- The run-times of generated JavaScript programs are in pair with the run-times of the corresponding binaries, meaning that the run-times of binaries and JavaScript programs of certain benchmarks have the same order. There are only three benchmarks breaking this rule (Braun, Knuth-Bendix, SumPuz), but one of these results (Braun) is related to the unavoidable speed loss coming from the optimization for stack use instead of speed.

- The high stack utilization of binaries clearly indicates high stack utilization of the related JavaScript programs.

- The high memory consumption of binaries clearly indicates high memory consumption of the related JavaScript programs.

According to these observations, the results indicate that the introduced STG $\rightarrow$ SAPL $\rightarrow$ JavaScript transformation preserve the properties of the original programs at a deep level. Notice, however, that the generated JavaScript benchmarks run about 1 to 2 magnitudes slower than their binary counterparts. Why should we translate Haskell/Clean programs to JavaScript then? The answers are manifold. First of all, there are tasks that inevitably must run on client side, like validation or management of GUI elements. For a second answer, one has to notice that running benchmarks is not part of the daily routine and most *normal* programs have acceptable run-times. This is even true for relatively large applications, like the SAPL to JavaScript compiler itself.

8. Related work

Compilation of traditional programming languages to JavaScript has drawn much attention in the last years as client-side processing for Internet applications was gaining importance. Virtually every modern language has some kind of technology which allows its client-side execution. A comprehensive overview of JavaScript related technologies of functional languages is given in [6]. Now, we are particularly interested in Haskell based compiler technologies.

There are several implementations of compilers from Haskell to JavaScript, although none of them has reached a full release stage as of the time of writing. Currently three important JavaScript backends exist for the various Haskell implementations. One of the first attempts to compile Haskell to JavaScript took a similar approach as ours; it also used STG as source language, but they implemented an STG machine in JavaScript [3]. Later on this project moved to YHC Core instead of STG, but it is abandoned since the development of York Haskell Compiler is canceled. There is another backend under development for Utrecht Haskell Compiler, which is going to be part of the upcoming release [4]. As for GHC, after some initial attempts, the most promising technology is GHCJS [1], which is still in its alpha stage at the time of writing. Finally, a bit different but interesting approach was chosen by JSHC [2], which implements a Haskell2010 compiler in JavaScript. Our approach, to convert the source language to obtain a new target platform is a novel idea, which has the definite advantage of reusing well established technologies and allows interoperability between different languages.

9. Conclusion and future work

In this paper we presented a technique to translate STG, one of the core languages of GHC, into SAPL, one of the core languages of Clean. To achieve this we used a method called compilation by transformation; the actual transformation consists of a series of rewrite rules. The presented transformation currently has a restriction to pure functions only. This is due to the lack of investigation of the impact of rewrite rules to monadic I/O. The viability of the technique is proved by a prototype implementation which was used for the compilation of an extensive range of benchmark programs. The translated benchmarks were converted to JavaScript to compare their run-times to the same benchmarks generated by the Clean complier. Analyzing the results we concluded that the transformation preserves well the run-time characteristics of the original programs.

We have not discussed the question of the reverse transformation, converting SAPL into STG. This task certainly would be harder, because STG, in contrast to SAPL, contains much information for the code generator. However, the necessary information is generated in a later stage by the SAPL to JavaScript compiler, thus the reverse conversation is probably possible.

As for the future work, the current prototype implementation of the compiler needs further improvement to be applicable for real world tasks. The most important modifications include the implementation of the data constructor grouping algorithm and completing the implementation of the full set of primitive functions of STG. Furthermore, cores of other Haskell compilers, e.g. YHC, can be considered to be transformed into SAPL using a similar technique which was presented in this paper.

References

[1] GHCJS `https://github.com/sviperll/ghcjs`

[2] JSCH `http://jshc.insella.se/`

[3] STG in JavaScript `http://www.haskell.org/haskellwiki/STG_in_Javascript`

[4] **Dijkstra, L.,** Haskell to JavaScript backend `http://utrechthaskellcompiler.wordpress.com/2010/10/18/haskell-to-javascript-backend/`

[5] **Domoszlai, L. and R. Plasmeijer,** SAPL/STG home page, `http://people.inf.elte.hu/dlacko/saplstg/`

[6] **Domoszlai, L., E. Bruël and J. M Jansen,** Implementing a non-strict functional language in JavaScript, *Acta Universitatis Sapientiae* 3(1):76–98, 2011, Sapienta University, Scienta Publishing House

[7] **Flanagan, C., A. Sabry, B. F. Duba and M, Felleisen,** The essence of compiling with continuations, in: *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation* (PLDI '93), New York, NY, USA, 1993, 237–247.

[8] **van Groningen, J., T. van Noort, P. Achten, P. Koopman and R. Plasmeijer,** Exchanging sources between Clean and Haskell - A Double-Edged Front End for the Clean Compiler, in: Jeremy Gibbons (Ed.) *Proceedings of the 2010 ACM SIGPLAN Haskell Symposium*, Baltimore, Maryland, USA, September 30, 2010, ACM, 49–60.

[9] **Jansen, J. M., P. Koopman and R. Plasmeijer,** Efficient interpretation by transforming data types and patterns to functions, in: Nilsson, H. (Ed.) *Proceedings Seventh Symposium on Trends in Functional Programming* (TFP 2006), Nottingham, UK, 19-21 April 2006, The University of Nottingham, 157–172.

[10] **Naylor, M. and C. Runciman,** The reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA, in: *Implementation and Application of Functional Languages* (IFP 2008), Hatfield, UK, September 10-12 2008, 129–146.

[11] **Naylor, M. and C. Runciman,** The reduceron reconfigured, in: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming* (ICFP '10), New York, NY, USA, 2010, 75–86.

[12] **Naylor, M., C. Runciman and J. Reich,** Reduceron home page, `http://www.cs.york.ac.uk/fp/reduceron/`

[13] **Peyton Jones, S.L.,** *The Implementation of Functional Programming Languages*, Prentice-Hall, Upper Saddle River, NJ, USA, 1987.

[14] **Peyton Jones, S.L.,** Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine - Version 2.5, *Journal of Functional Programming*, **2** (1992), 127–202.

[15] **Peyton Jones, S.L., C. Hall, K. Hammond, J. Cordy, H. Kevin, W. Partain and P. Wadler,** The Glasgow Haskell compiler: a technical overview, in: *Proc. UK Joint Framework for Information Technology Technical Conference* (JFIT), (1993), 249–257.

L. Domoszlai
Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
Pázmány P. sétány 1/C
H-1117 Budapest, Hungary
`dlacko@pnyf.inf.elte.hu`

R. Plasmeijer
Radboud University
Software Technology Research Group
Nijmegen, The Netherlands
`rinus@cs.ru.nl`