Realising nondeterministic I/O in the Glasgow Haskell Compiler

Technical Report Frank-17

David Sabel

Institut für Informatik
Johann Wolfgang Goethe-Universität
Frankfurt, Germany
Email: sabel@informatik.uni-frankfurt.de

December 3, 2003

Abstract

In this paper we demonstrate how to relate the semantics given by the nondeterministic call-by-need calculus FUNDIO [SS03] to Haskell. After introducing new correct program transformations for FUNDIO, we translate the core language used in the Glasgow Haskell Compiler into the FUNDIO language, where the IO construct of FUNDIO corresponds to direct-call IO-actions in Haskell. We sketch the investigations of [Sab03b] where a lot of program transformations performed by the compiler have been shown to be correct w.r.t. the FUNDIO semantics. This enabled us to achieve a FUNDIO-compatible Haskell-compiler, by turning off not yet investigated transformations and the small set of incompatible transformations. With this compiler, Haskell programs which use the extension `unsafePerformIO` in arbitrary contexts, can be compiled in a 'safe' manner.

Contents

1 Introduction

This paper gives a summary of the work in [Sab03b] which is based upon the FUNDIO calculus [SS03]. The nondeterministic call-by-need calculus FUNDIO provides an IO-interface which can be used to model direct-call IO within Haskell, i.e. the IO-actions need no special treatment like monads. The language is no longer pure in the usual meaning, but [SS03] defines a contextual equivalence for FUNDIO, which enables us to compare programs and to substitute a term with a contextual equivalent expression. The Haskell extension `unsafePerformIO` makes an easy implementation of direct-call IO possible, i.e. a direct-call IO-action can be built by applying `unsafePerformIO` to a monadic IO-action. For example, a direct-call IO-action `dPutChar` which prints a character to the standard output can be defined by using the monadic function `putChar`:

```
dPutChar :: Char -> ()
dPutChar c = unsafePerformIO (putChar c)
```

But the use of `unsafePerformIO` in existing compilers is limited to special cases, i.e. `unsafePerformIO` should only be used to implement a function if the function can also be implemented by using conventional methods[1]. The criteria for safe uses of `unsafePerformIO` are not formally specified and are frequently discussed on several Haskell-related mailing lists. Our experiences show that programs, which use `unsafePerformIO` in arbitrary contexts, i.e. these uses are unsafe in the usual sense, show up different IO-behavior when compiling with different levels of optimisation. So our aim is to perform only such optimisations which are compatible with the FUNDIO semantics. The calculus does not specify the order of evaluating IO-actions, i.e. to sequentialize the execution of IO-actions special provisions must be made. But FUNDIO specifies how often IO-actions are evaluated, i.e. it only allows permutations of IO-actions. So, programs with different numbers of IO-actions are never contextually equivalent in FUNDIO.

We have implemented the results in the Glasgow Haskell Compiler (GHC), by turning off FUNDIO-incompatible program transformations and those that have not yet been investigated, and achieved a modification of the compiler which is called HasFuse [Sab03a].

1.1 Overview

In Section 2 we present the FUNDIO-calculus as defined in [SS03], which is a nondeterministic call-by-need lambda-calculus, where the nondeterminism is used to model an IO-interface. We present a contextual preorder and equivalence, which is then used to define the correctness of a program transformation. After describing a large set of correct program transformations of [SS03] we extend this set by introducing some new transformations. In Section 3 we present a translation from the core language used in the Glasgow Haskell Compiler to the FUNDIO language. Based on this translation we define correctness for program transformations performed in the GHC. Then we investigate a lot of program transformations regarding the defined correctness. In the latter sections we summarize the work and suggest directions for further work.

[1] For example, [The03, Chapter 13] gives some hints when it is safe to use `unsafePerformIO`.

2 The FUNDIO calculus

In this section we give an overview of the FUNDIO-language and its corresponding reduction rules. After that, a contextual preorder is presented, which is used to define a *correct program transformation.* The section ends by presenting a large set of program transformations, which have been shown to be correct in [SS03] and [Sab03b].

2.1 Syntax

We define the FUNDIO language similarly to [SS03]:

Definition 2.1. (L_{FUNDIO}) *We assume there is a finite set $\mathcal{C}$ of constructors with $|\mathcal{C}| = N \geq 2$. The constructors are numbered where c_i denotes the i-th constructor. The constructor c_N is the special constant* `lambda`*, which can only occur as a pattern in a* `case` *alternative. With $ar(c_i)$ we denote the arity of constructor c_i. Figure 1 presents the language L_{FUNDIO}. Valid expressions can be derived starting with the nonterminal* **E***, where the following conditions must hold: The alternatives of a* `case` *expression are complete, i.e. for every constructor $c \in \mathcal{C}$ there is exactly one alternative. The variables V_i in a* `letrec` *expression or a* `case` *pattern are distinct and the order of the bindings in a* `letrec` *environment is commutable, i.e. we do not distinguish expressions with commuted bindings.* `letrec` *expressions with an empty set of bindings are allowed, e.g.* ($\texttt{letrec}\ \{\}\ \texttt{in}\ s$) *is a valid expression (if s is valid).*

$$
\begin{array}{lcll}
\mathbf{E} & ::= & \mathrm{V} & \text{(variable)} \\
 & \mid & (c_i\ \mathbf{E}_1 \ldots \mathbf{E}_{ar(c)}) & \text{(constructor application)} \\
 & \mid & (\texttt{IO}\ \mathbf{E}) & \text{(IO expression)} \\
 & \mid & (\texttt{case}\ \mathbf{E}\ \mathbf{Alt}_1 \ldots \mathbf{Alt}_N) & \text{(case expression)} \\
 & \mid & (\mathbf{E}_1\ \mathbf{E}_2) & \text{(application)} \\
 & \mid & (\lambda \mathrm{V}.\mathbf{E}) & \text{(abstraction)} \\
 & \mid & (\texttt{letrec}\ \mathrm{V}_1 = \mathbf{E}_1, \ldots, \mathrm{V}_n = \mathbf{E}_n\ \texttt{in}\ \mathbf{E}) & \text{(letrec expression)} \\
\mathbf{Alt} & ::= & (\mathbf{Pat} \rightarrow \mathbf{E}) & \text{(alternative)} \\
\mathbf{Pat} & ::= & (c_j\ \mathrm{V}_1 \ldots \mathrm{V}_{ar(c_j)}) & \text{(pattern)}
\end{array}
$$

where $i \in \{1, \ldots, N-1\}$ and $j \in \{1, \ldots N\}$.

Figure 1: L_{FUNDIO} - The FUNDIO language

Convention 2.2. *We use the following notation to abbreviate some expressions.*

- *Instead of* ($\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ t$)*, we also write* ($\texttt{letrec}\ Env\ \texttt{in}\ t$)*.*

- *Instead of* ($\texttt{case}$ s Alt_1 $\ldots$ Alt_n)*, we also write* ($\texttt{case}$ s $Alts$)*.*

- *If the meaning is clear, we omit parenthesis. The application is left-associative, i.e.* $(a_1 \ \ldots \ a_n)$ *is an abbreviation for* $(\ldots((a_1\ a_2)\ldots)\ a_n)$*.*

- *Instead of* $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.s))\ldots)$*, we also use* $(\lambda x_1 \ldots x_n.s)$*.*

In the following we use *free* and *bound* variables and the disjoint variable convention as well as *open* and *closed* terms. The definitions for the FUNDIO calculus can be found in [SS03, Sab03b].

2.2 Contexts

A *context* is an expression with a hole in it. We represent the hole by the symbol $[\cdot]$.

Definition 2.3. (Context) *A context C is defined by the following grammar.*

$$\begin{array}{lcl} C & ::= & [\cdot] \mid (\lambda x.C) \mid (C\ E) \mid (E\ C) \mid (\texttt{IO}\ C) \mid (c\ E\ \ldots\ E\ C\ E\ \ldots\ E) \\ & \mid & (\texttt{case}\ C\ Alts) \mid (\texttt{case}\ E\ Alt_1 \ldots (Pat \to C) \ldots Alt_n) \\ & \mid & (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ C) \\ & \mid & (\texttt{letrec}\ x_1 = E_1, \ldots, x_{i-1} = E_{i-1}, x_i = C, x_{i+1} = E_{i+1}, \ldots x_n = E_n\ \texttt{in}\ E) \end{array}$$

If D is a context, then we denote $D[t]$ as the expression which arises by placing t instead of the hole in D. *Reduction contexts* are those contexts, in which we will perform (especially normal order) reductions:

Definition 2.4. (Reduction context) *The class R of reduction contexts is built upon the subclass R^- of* weak reduction contexts*. Both classes are defined by the following grammar:*

$$\begin{array}{lcl} R^- & ::= & [\cdot] \mid (R^-\ E) \mid (\texttt{case}\ R^-\ Alts) \mid (\texttt{IO}\ R^-) \\ R & ::= & R^- \mid (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ R^-) \\ & \mid & (\texttt{letrec}\ x_1 = R_1^-, \ldots, x_j = R_j^-[x_{j-1}], \ldots\ \texttt{in}\ R^-[x_j]) \\ & & \textit{where } R^-, R_i^- \textit{ are contexts of class } R^-. \end{array}$$

Another context class are the *surface contexts*. These contexts do not have a hole in the body of an abstraction.

Definition 2.5. (Surface context) *A surface context S is defined by the following grammar:*

$$\begin{array}{lcl} S & ::= & [\cdot] \mid (S\ E) \mid (E\ S) \mid (\texttt{IO}\ S) \mid (c\ E\ \ldots\ E\ S\ E\ \ldots\ E) \mid (\texttt{case}\ S\ Alts) \\ & \mid & (\texttt{case}\ E\ Alt_1 \ldots (Pat \to S) \ldots Alt_n) \mid (\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ S) \\ & \mid & (\texttt{letrec}\ x_1 = E_1, \ldots, x_{i-1} = E_{i-1}, x_i = S, x_{i+1} = E_{i+1}, \ldots x_n = E_n\ \texttt{in}\ E) \end{array}$$

2.3 Reduction rules

The following definition is similar to [SS03] and presents the reduction rules of the FUNDIO calculus.

Definition 2.6. (Reduction rules) *Figures 2 and 3 define the* reduction rules. *A rule*

$$\textit{(name)} \qquad a \longrightarrow b$$

has the following meaning: An expression of form a can be replaced by an expression of form b by using the rule (name).

We denote the union of (cp-in) and (cp-e) with (cp), the union of (llet-in) and (llet-e) with (llet), the union of (case-c), (case-in), (case-e) and (case-lam) with (case) and the union of (IOr-c), (IOr-in) and (IOr-e) with (IOr). Similar to [SS03] we define the reduction (lll) as the union of (llet), (lapp), (lcase) and (IOlet).

If necessary, we label the reduction with the used rule and/or with the context, where the reduction takes place, e.g. $\xrightarrow{R,case}$ is a (case)-reduction inside a reduction context. We denote the transitive closure of a reduction with the symbol $+$, the reflexive-transitive closure with $*$. For example, $\xrightarrow{(llet)^+}$ is the transitive closure of $\xrightarrow{llet}$. Note that the (IOr) reduction is nondeterministic, it models IO-actions in the following way: If $(\texttt{IO}\ c) \xrightarrow{IOr} d$, then after outputting the *output value* c, the *input value* d is obtained nondeterministically. The idea is that the user inputs the input value, so the program does not know, what the result of the (IOr) reduction is.

Instead of defining the *normal order reduction* $\xrightarrow{n}$ of the FUNDIO calculus explicitly, we refer to [SS03] and make some remarks about it. The *normal order redex* of a term t is the subexpression on which the normal order reduction (i.e. one of the reduction rules of Definition 2.6) is applied. [SS03, Lemma 5.4] shows that for all terms $t \in L_{FUNDIO}$ holds:

- If t has a normal order redex, then this redex is unique.
- If the normal order reduction of t is a deterministic reduction rule (i.e. not an (IOr) reduction), then the normal order reduction is unique.
- If the normal order reduction of t is an (IOr) reduction and the IO-pair of the reduction is given, then the normal order reduction is unique.

Because FUNDIO is a call-by-need calculus, the normal order reduction respects sharing. In contrast to [AFM+95] in the expression $((\texttt{letrec}\ x = (\texttt{letrec}\ y = s_y\ \texttt{in}\ y)\ \texttt{in}\ x)\ t)$ the normal order reduction of FUNDIO does firstly a (lapp) reduction before adjusting the environment with a (llet) reduction. We give another example of reducing a term by normal order reductions:

(lbeta) $((\lambda x.\ s)\ t) \longrightarrow (\mathtt{letrec}\ x = t\ \mathtt{in}\ s)$

(cp-in) $(\mathtt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, Env\ \ \mathtt{in}\ C[x_j])$
$\longrightarrow (\mathtt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, Env\ \mathtt{in}\ C[s_1])$
where s_1 is an abstraction

(cp-e) $(\mathtt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, x_{j+1} = C[x_j], Env\ \ \mathtt{in}\ s)$
$\longrightarrow (\mathtt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_j = x_{j-1}, x_{j+1} = C[s_1], Env\ \mathtt{in}\ s)$
where s_1 is an abstraction

(llet-in) $(\mathtt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \mathtt{in}\ (\mathtt{letrec}\ \ y_1 = t_1, \ldots, y_m = t_m\ \mathtt{in}\ r))$
$\longrightarrow (\mathtt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, y_1 = t_1, \ldots, y_m = t_m\ \mathtt{in}\ r)$

(llet-e) $(\mathtt{letrec}\ x_1 = s_1, \ldots,$
$x_i = (\mathtt{letrec}\ y_1 = t_1, \ldots, y_m = t_m\ \mathtt{in}\ s_i), \ldots,$
$x_n = s_n$
$\mathtt{in}\ r)$
$\rightarrow (\mathtt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, y_1 = t_1, \ldots, y_m = t_m\ \mathtt{in}\ r)$

(lapp) $((\mathtt{letrec}\ Env\ \mathtt{in}\ t)\ s) \longrightarrow (\mathtt{letrec}\ Env\ \mathtt{in}\ (t\ s))$

(lcase) $(\mathtt{case}\ (\mathtt{letrec}\ Env\ \mathtt{in}\ t)\ Alts) \longrightarrow (\mathtt{letrec}\ \ Env\ \mathtt{in}\ (\mathtt{case}\ t\ Alts))$

(case-c) $(\mathtt{case}\ (c_i\ t_1 \ldots t_n)\ \ldots\ ((c_i\ y_1 \ldots y_n) \rightarrow t) \ldots)$
$\longrightarrow (\mathtt{letrec}\ y_1 = t_1, \ldots, y_n = t_n\ \mathtt{in}\ t)$

(case-lam) $(\mathtt{case}\ (\lambda x.\ s)\ \ldots\ (\mathtt{lambda}\!\rightarrow t) \ldots) \longrightarrow\ (\mathtt{letrec}\ \{\}\ \mathtt{in}\ t)$

(case-in) $(\mathtt{letrec}\ x_1 = (c_i\ t_1\ \ldots\ t_n), x_2 = x_1, \ldots, x_m = x_{m-1}, \ldots$
$\mathtt{in}\ C[\mathtt{case}\ x_m\ \ldots((c_i\ z_1\ \ldots\ z_n) \rightarrow t)])$
$\longrightarrow (\mathtt{letrec}\ x_1 = (c_i\ y_1\ \ldots\ y_n), y_1 = t_1, \ldots, y_n = t_n$
$x_2 = x_1, \ldots, x_m = x_{m-1}, \ldots$
$\mathtt{in}\ C[(\mathtt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \mathtt{in}\ t)])$
where the y_i are fresh variables

(case-e) $(\mathtt{letrec}\ x_1 = (c_i\ t_1\ \ldots\ t_n), x_2 = x_1, \ldots, x_m = x_{m-1}, \ldots$
$u = C[\mathtt{case}\ x_m\ \ldots((c_i\ z_1\ \ldots\ z_n) \rightarrow r_1)]$
$\mathtt{in}\ r_2)$
$\longrightarrow (\mathtt{letrec}\ x_1 = (c_i\ t_1\ \ldots\ t_n),$
$y_1 = t_1, \ldots, y_n = t_n,$
$x_2 = x_1, \ldots, x_m = x_{m-1}, \ldots$
$u = C[(\mathtt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \mathtt{in}\ r_1)]$
$\mathtt{in}\ r_2)$
where the y_i are fresh variables

Figure 2: Reduction rules of the FUNDIO calculus

(IOlet) $(\texttt{IO}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)) \longrightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{IO}\ s))$

In the following three rules c and d are constants and (c, d) is the *IO-pair* of the reduction.

(IOr-c) $(\texttt{IO}\ c) \longrightarrow d$

(IOr-in) $(\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ C[(\texttt{IO}\ x_m)])$
$\longrightarrow (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ C[d])$

(IOr-e) $(\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, u = C[(\texttt{IO}\ x_m)], Env\ \texttt{in}\ r)$
$\longrightarrow (\texttt{letrec}\ x_1 = c, x_2 = x_1, \ldots, x_m = x_{m-1}, u = C[d], Env\ \texttt{in}\ r)$

Figure 3: IO reduction rules of the FUNDIO calculus

Example 2.7. *We reduce the following expression t in normal order. Let $c, d \in \mathcal{C}$ be constants.*

$$\begin{array}{rl}
t = & (\texttt{letrec}\ x_1 = ((\lambda y.y)\ c), x_2 = x_1, x_3 = (\texttt{case}\ x_2\ \ldots(c \to c)\ldots)\ \texttt{in}\ (\texttt{IO}\ x_3)) \\
\xrightarrow{n,lbeta} & (\texttt{letrec}\ x_1 = (\texttt{letrec}\ y = c\ \texttt{in}\ y), x_2 = x_1, x_3 = (\texttt{case}\ x_2\ \ldots(c \to c)\ldots) \\
& \texttt{in}\ (\texttt{IO}\ x_3)) \\
\xrightarrow{n,llet-e} & (\texttt{letrec}\ x_1 = y, y = c, x_2 = x_1, x_3 = (\texttt{case}\ x_2\ \ldots(c \to c)\ldots)\ \texttt{in}\ (\texttt{IO}\ x_3)) \\
\xrightarrow{n,case-e} & (\texttt{letrec}\ x_1 = y, y = c, x_2 = x_1, x_3 = (\texttt{letrec}\ \{\}\ \texttt{in}\ c)\ \texttt{in}\ (\texttt{IO}\ x_3)) \\
\xrightarrow{n,llet-e} & (\texttt{letrec}\ x_1 = y, y = c, x_2 = x_1, x_3 = c\ \texttt{in}\ (\texttt{IO}\ x_3)) \\
\xrightarrow{n,IOr-in} & (\texttt{letrec}\ x_1 = y, y = c, x_2 = x_1, x_3 = c\ \texttt{in}\ d) \\
& \textit{No further normal order reduction is applicable.}
\end{array}$$

We now define *values* and *WHNF*s:

Definition 2.8. (Value and WHNF) *A* value *is a constructor application or an abstraction. A* weak head normal form (WHNF) *is*

- *a value, or*
- *an expression of the form* $(\texttt{letrec}\ Env\ \texttt{in}\ t)$*, where t is a value, or*
- *an expression of the form* $(\texttt{letrec}\ x_1 = (c\ t_1\ \ldots\ t_{ar(c)}), x_2 = x_1, \ldots, x_m = x_{m-1}, Env\ \texttt{in}\ x_m)$.

The last expression of example 2.7 where no rule is applicable is a WHNF, because d is a value. Note that a WHNF has no normal order reduction.

Definition 2.9. (bot-term) *Let t be a closed expression. We say t is a* bot-term, *if t has no normal order reduction, that ends with a WHNF.*

[SS03] shows that all bot-terms are contextually equivalent and that their equivalence class is the least element of the contextual preorder.

2.4 Contextual equivalence

A *reduction sequence* $s_1 \to \ldots \to s_n$ is a sequence of reductions. If not otherwise specified, these are reductions of the FUNDIO calculus. We call a reduction sequence starting with an expression t, that consists only of normal order reductions as the *NO-reduction sequence of* t. In the following we firstly define IO-multisets, IO-sequences and termination and finally the contextual equivalence is defined.

2.4.1 IO-multisets and IO-sequences

Definition 2.10. (IO-pairs, IO-multisets and IO-sequences) *An* IO-pair *is a pair* (a, b)*, where* a *and* b *are constants of* $\mathcal{C}$*:*

- *The IO-pair of an* (IOr) *reduction is the pair* (c, d) *consisting of the output and input value as defined in figure 3.*
- *Reductions of type* a *with* $a \notin \{(\text{IOr-c}), (\text{IOr-in}), (\text{IOr-e})\}$ *do not have an IO-pair.*

An IO-sequence *is a finite sequence of IO-pairs. The IO-sequence* $IOS(s_1 \to \ldots \to s_n)$ *of a reduction sequence* $s_1 \to \ldots \to s_n$ *is defined as follows:*

- *If* $s_1 \to s_2$ *is an* (IOr) *reduction with IO-pair* (a, b)*, then* $IOS(s_1 \to \ldots \to s_n) := (a, b), IOS(s_2 \to \ldots \to s_n)$.
- *If* $s_1 \to s_2$ *is not an* (IOr) *reduction, then* $IOS(s_1 \to \ldots \to s_n) := IOS(s_2 \to \ldots \to s_n)$.

An IO-multiset *is a finite set of IO-pairs. The IO-multiset* $IOM(s_1 \to \ldots \to s_n)$ *of the reduction sequence* $s_1 \to \ldots \to s_n$ *is the multiset consisting of the elements of* $IOS(s_1 \to \ldots \to s_n)$.

2.4.2 Termination

Definition 2.11. *Let* t *be an expression and* P *be a finite IO-multiset. We write* $t{\Downarrow}(P)$ *if there is a NO-reduction sequence* Q *of* t*, that ends with a WHNF and* $IOM(Q) = P$*. Then we say* t terminates for the IO-multiset P*.*

For a closed term t*, we say* t *has a* bot-reduction *iff there is a normal order reduction* $t \xrightarrow{n,*} t'$ *where* t' *is a bot-term. If* t *has a bot-reduction, we write* $t{\Uparrow}$*.*

Example 2.12. *Let $c, d, e \in \mathcal{C}$ be constants and $\bot$ be a bot-term. Let $t \in L_{FUNDIO}$ be the following expression:*

$$t := (\texttt{case}\ (\texttt{IO}\ c)\ (d \to \bot)\ (e \to e)\ \ldots)$$

Then the following holds:

- $t\Uparrow$, *since the normal order reduction* $t \xrightarrow{n,IOr,(c,d)} (\texttt{letrec}\ \{\}\ \texttt{in}\ \bot)$ *ends with a bot-term.*
- $t \xrightarrow{n,IOr,(c,e)} (\texttt{letrec}\ \{\}\ \texttt{in}\ e)$ *is a normal order reduction of t that ends with a WHNF. So, t is not a bot-term.*
- *Let $P = \{(c, e)\}$, then $t\Downarrow(P)$.*

2.4.3 Contextual equivalence

Definition 2.13. (Contextual preorder and equivalence) *The* contextual preorder *$\leq_c$ on terms s, t is the following binary relation:*

$$s \leq_c t \textit{ iff } \forall C[\cdot] : \big((\forall P : C[s]\Downarrow(P) \implies C[t]\Downarrow(P)) \wedge (C[t]\Uparrow \implies C[s]\Uparrow)\big)$$

The contextual equivalence *$\sim_c$ on terms s, t is the binary relation with*

$$s \sim_c t \textit{ iff } s \leq_c t \wedge t \leq_c s$$

A *precongruence* is a preorder $\preceq$ on terms, with $s \preceq t \implies C[s] \preceq C[t]$ for all contexts C. A *congruence* is a precongruence which is also an equivalence relation. [SS03, Proposition 6.7] shows: $\leq_c$ is a precongruence and $\sim_c$ is a congruence.

2.5 Program transformations

Definition 2.14. (Correct program transformation) *A* program transformation *is a binary relation on expressions. A program transformation T is* correct *if for all expressions $s_1, s_2 \in L_{FUNDIO}$ holds: $s_1\ T\ s_2 \implies s_1 \sim_c s_2$.*

In [SS03, Theorem 16.1 and Proposition 16.2] has been proven that all deterministic reduction rules (namely (lbeta), (lapp), (llet), (lcase), (IOlet), (cp), (case)) are correct program transformations and that the rules (IOr-c), (IOr-in) and (IOr-e) are not correct program transformations if $|\mathcal{C}| \geq 2$.

Figure 4 defines further program transformations, which have been proven to be correct in [SS03], where we use the following unions: We denote the union of (gc-1) and (gc-2) with (gc), the union of (cpx-in) and (cpx-e) with (cpx), the union of (cpcx-in) and (cpcx-e) with (cpcx) and finally we denote the union of (ucp-1) and (ucp-2) with (ucp).

Garbage Collection

(gc-1) $(\mathtt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, Env\ \mathtt{in}\ t) \longrightarrow (\mathtt{letrec}\ Env\ \mathtt{in}\ t)$
if for all $i : x_i$ does not occur in Env nor in t.

(gc-2) $(\mathtt{letrec}\ \{\}\ \mathtt{in}\ t) \longrightarrow t$

Copying variables

(cpx-in) $(\mathtt{letrec}\ x = y, Env\ \mathtt{in}\ \ C[x]) \longrightarrow (\mathtt{letrec}\ x = y, Env\ \ \mathtt{in}\ \ C[y])$
where y is a variable and $x \neq y$.

(cpx-e) $(\mathtt{letrec}\ x = y, z = C[x], Env\ \mathtt{in}\ \ t)$
$\longrightarrow (\mathtt{letrec}\ x = y, z = C[y], Env\ \mathtt{in}\ \ t)$
where y is a variable and $x \neq y$.

Copying constructors

(cpcx-in) $(\mathtt{letrec}\ x_1 = c\ t_1 \ldots t_m, Env\ \mathtt{in}\ \ C[x])$
$\longrightarrow (\mathtt{letrec}\ x_1 = c\ y_1 \ldots y_m,$
$y_1 = t_1, \ldots, y_m = t_m, Env\ \mathtt{in}\ \ C[c\ y_1 \ldots y_m])$

(cpcx-e) $(\mathtt{letrec}\ x_1 = c\ t_1 \ldots t_m, z = C[x], Env\ \mathtt{in}\ t)$
$\longrightarrow (\mathtt{letrec}\ x_1 = c\ y_1 \ldots y_m,$
$y_1 = t_1, \ldots, y_m = t_m, z = C[c\ y_1 \ldots y_m], Env\ \ \mathtt{in}\ t)$

Lambda lifting

(llift) $C[s[z]] \longrightarrow C[(\lambda x.\ s[x])\ z]$, where z is a Variable

Copying unique expressions

(ucp-1) $(\mathtt{letrec}\ x = s, Env\ \mathtt{in}\ \ S[x]) \longrightarrow (\mathtt{letrec}\ Env\ \mathtt{in}\ \ S[s])$
if x occurs exactly once in $Env, S[x]$ and does not occur in s.

(ucp-2) $(\mathtt{letrec}\ x = s, Env, y = S[x]\ \mathtt{in}\ \ t)$
$\longrightarrow (\mathtt{letrec}\ Env, y = S[s]\ \mathtt{in}\ \ t)$
if x occurs exactly once in $Env, S[x], t$ and does not occur in s.

Other transformations

(xch) $(\mathtt{letrec}\ x = t, y = x, Env\ \mathtt{in}\ r) \longrightarrow (\mathtt{letrec}\ y = t, x = y, Env\ \mathtt{in}\ r)$

(betavar) $C[(\lambda x.\ s)\ y] \longrightarrow C[s[y/x]]$, if y is a variable.

Figure 4: Further program transformations of [SS03]

The next lemma presents another result of [SS03] about bot-terms, which we will use in later sections.

Lemma 2.15. *Let $\perp \in L_{FUNDIO}$ be a bot-term. Then for all reduction contexts R: $R[\perp] \sim_c \perp$.*

Proof. See [SS03, Corollary 20.18]. $\square$

2.6 Transformations on case expressions

Definition 2.16. *Figure 5 defines some new program transformations, which all operate on* `case` *expressions.*

With rule (capp) applications to `case` expressions can be shifted inside the alternatives. The (ccpcx) rule allows to copy patterns into a right hand side of a `case` alternative if the scrutinee is a variable. The rule (lcshift) shifts outer bindings into `case` alternatives, where the expression must have a special form. The rule is necessary for proving the (ccase-in) rule. The (ccase) rule can be applied to nested `case` expressions and commutes the order of the case expressions. The (ccase-in) rule is a special variant of the (ccase) rule. The (crpl) rule allows to replace a right hand side of a `case` alternative if the alternative is not reachable by reduction. In [Sab03b] we have shown that all of these `case` transformations are correct program transformations. For the proofs of (capp), (ccpcx), (ccase) and (crpl) we used the technique of complete sets of commuting and forking diagrams together with the so-called context lemma of [SS03][2]. The remaining transformations can be shown to be correct by transforming their left hand sides into their right hand sides, by using only correct program transformations

2.7 Transformations for copying expressions

In [SS03] some transformations for copying specific expressions into specific contexts have already been defined and proven to be correct. Variables ((cpx) rule), constants ((cpcx) rule) and abstractions ((cp) rule) can be copied into arbitrary contexts. Furthermore, the rule (ucp) has been shown to be correct, hence it is allowed to copy expressions if they occur once and not in a body of an abstraction. Below we define further transformations, which allow (restricted) copying.

Definition 2.17. **(L_{cheap})** *Let $L_{cheap} \subset L_{FUNDIO}$ be the language defined by the following grammar:*

$$
\begin{array}{llll}
\mathbf{E}_c & ::= & V & \textit{variable} \\
 & \mid & (\lambda V.s) & \textit{where } s \in L_{FUNDIO} \\
 & \mid & (c_i\ \mathbf{E}_{c,1}\ \ldots \mathbf{E}_{c,n}) & \textit{where } ar(c_i) = n \\
 & \mid & (\lambda x_1 \ldots x_n.(c_i\ x_1 \ldots x_n))\ \ \mathbf{E}_{c,1}\ \ldots\ \ \mathbf{E}_{c,m} & \textit{where } ar(c_i) = n + m
\end{array}
$$

[2] The technique and the context lemma are described in detail in [SS03] and [Sab03b].

(capp) $((\texttt{case}\ s\ (p_1 \rightarrow t_1)\ \ldots\ (p_N \rightarrow t_N))\ t)$
$\longrightarrow (\texttt{case}\ s\ (p_1 \rightarrow (t_1\ t))\ \ldots\ (p_N \rightarrow (t_N\ t)))$

(ccpcx) $(\texttt{case}\ x\ (p_1 \rightarrow t_1) \ldots ((c_i\ y_1 \ldots y_{ar(c_i)}) \rightarrow C[x]) \ldots (p_N \rightarrow t_N))$
$\longrightarrow (\texttt{case}\ x$
$(p_1 \rightarrow t_1) \ldots$
$((c_i\ y_1 \ldots y_{ar(c_i)}) \rightarrow C[(c_i\ y_1 \ldots y_{ar(c_i)})]) \ldots$
$(p_N \rightarrow t_N))$
where x is a variable and $1 \leq i < N$.

(lcshift) $(\texttt{letrec}\ y = s, Env\ \texttt{in}\ R^-[(\texttt{case}\ y\ (p_1 \rightarrow t_1) \ldots (p_N \rightarrow t_N))])$
$\longrightarrow (\texttt{letrec}\ Env\ \texttt{in}\ R^-[(\texttt{case}\ s$
$(p_1 \rightarrow (\texttt{letrec}\ y = p_1\ \texttt{in}\ t_1))$
$\ldots$
$(p_N \rightarrow (\texttt{letrec}\ y = p_N\ \texttt{in}\ t_N)))])$
if y does not occur free in s, Env and R^-.

(ccase) $(\texttt{case}\ (\texttt{case}\ s\ (p_1 \rightarrow t_1)\ \ldots\ (p_N \rightarrow t_N))\ Alts)$
$\longrightarrow (\texttt{case}\ s\ (p_1 \rightarrow (\texttt{case}\ t_1\ Alts))\ \ldots\ (p_N \rightarrow (\texttt{case}\ t_N\ Alts)))$

(ccase-in) $(\texttt{letrec}\ y = (\texttt{case}\ s\ (p_1 \rightarrow t_1) \ldots (p_N \rightarrow t_N))\ \texttt{in}\ (\texttt{case}\ y\ Alts))$
$\longrightarrow (\texttt{case}\ s$
$(p_1 \rightarrow (\texttt{letrec}\ y = t_1\ \texttt{in}\ (\texttt{case}\ y\ Alts)))$
$\ldots$
$(p_N \rightarrow (\texttt{letrec}\ y = t_N\ \texttt{in}\ (\texttt{case}\ y\ Alts))))$
if y does not occur free in $(\texttt{case}\ s\ (p_1 \rightarrow t_1) \ldots (p_N \rightarrow t_N))$.

(crpl) $(\texttt{case}\ s\ (p_1 \rightarrow t_1) \ldots ((c_i\ y_1 \ldots y_{ar(c_i)}) \rightarrow t_i) \ldots (p_N \rightarrow t_N))$
$\longrightarrow (\texttt{case}\ s\ (p_1 \rightarrow t_1) \ldots ((c_i\ y_1 \ldots y_{ar(c_i)}) \rightarrow q) \ldots (p_N \rightarrow t_N))$
can be applied in a context C if this context does not bind the free variables of s, so that s in C could be reduced to a constructor application $(c_i\ a_1 \ldots a_n)$. There $1 \leq i < N$ and q is a arbitrary closed expression.

Figure 5: `case` transformations

Definition 2.18. *Figure 6 defines the rules* (cpcheap-in)*,* (cpcheap-e)*,* (brcp-in)*,* (brcp-e)*,* (ucpb-in) *and* (ucpb-e)*. The union of* (cpcheap-in) *and* (cpcheap-e) *is denoted with* (cpcheap)*, the union of* (brcp-in) *and* (brcp-e) *with* (brcp) *and the union of* (ucpb-in) *and* (ucpb-e) *with* (ucpb)*.*

The rule (cpcheap) combines some (cp), (cpx) and (cpcx) reductions, so that expressions that are built only by variables, abstractions or constructor applications (with arguments of L_{cheap}) can be copied in one step. The last expression in the definition of L_{cheap} is necessary to simulate unsaturated constructor applications (which are not allowed in L_{FUNDIO}). The rule (brcp) allows to float outer `letrec` bindings into alternatives of `case` expressions and the rule is used for the proof of the (ucpb) rule, which is an extension of the (ucp) rule: expressions can be copied into a `case` alternative (if the occurrence is not in a body of an abstraction), also if the variable occurs more then once in other alternatives. In [Sab03b] we have shown that all of the copying transformations are correct. The correctness of the (cpcheap) rule can be proven by induction, where the base cases are correct, because of the (cpx), (cpcx) and (cp) rules. The (brcp) rule has been proven to be correct by using the technique of complete sets of commuting and forking diagrams. The (ucpb) rule can be shown to be correct by transforming the left hand side into the right hand side of the rule by using only correct program transformations, especially the (brcp) rule.

2.8 Strictness optimisation

In the following definition we introduce *strict abstractions.*

Definition 2.19. (Strict abstraction) *An abstraction s is* strict *if* $(s\ \bot) \sim_c \bot$*, where* $\bot$ *is a bot-term.*

Definition 2.20. *The rule (streval) is defined as follows*

$$\begin{array}{ll}
\text{(streval)} & ((\lambda y.s)\ t) \\
& \longrightarrow (\texttt{letrec}\ w = t\ \texttt{in} \\
& \qquad\quad (\texttt{case}\ w\ (pat_1 \to ((\lambda y.s)\ w)) \ldots (pat_N \to ((\lambda y.s)\ w)))) \\
& \text{if } (\lambda y.s) \text{ is a strict abstraction}
\end{array}$$

We yet do not have a proof of correctness for the (streval) transformation, but we conjecture that the transformation is correct.

2.9 Results

The following theorem summarizes that all introduced rules — except of the (streval) rule — are correct program transformations.

Theorem 2.21. *The rules (capp), (ccpcx), (lcshift), (ccase), (ccase-in), (crpl), (cpcheap), (brcp) und (ucpb) are correct program transformations.*

(cpcheap-in) $(\texttt{letrec}\ x = t,\ Env\ \texttt{in}\ C[x]) \longrightarrow (\texttt{letrec}\ x = t,\ Env\ \texttt{in}\ C[t])$
where $t \in L_{cheap}$

(cpcheap-e) $(\texttt{letrec}\ x = t,\ y = C[x],\ Env\ \texttt{in}\ s)$
$\longrightarrow (\texttt{letrec}\ x = t,\ y = C[t],\ Env\ \texttt{in}\ s)$
where $t \in L_{cheap}$

(brcp-in) $(\texttt{letrec}\ y = s,\ Env\ \texttt{in}\ R^-[(\texttt{case}\ t\ (pat_1 \to t_1) \ldots (pat_N \to t_N))])$
$\longrightarrow (\texttt{letrec}\ Env\ \texttt{in}$
$R^-[(\texttt{case}\ t\ (pat_1 \to (\texttt{letrec}\ y = s\ \texttt{in}\ t_1))$
$\ldots$
$(pat_N \to (\texttt{letrec}\ y = s\ \texttt{in}\ t_N)))])$
if y does not occur free in R^-, Env, s and t

(brcp-e) $(\texttt{letrec}$
$y = s, x = R^-[(\texttt{case}\ t\ (pat_1 \to t_1) \ldots (pat_N \to t_N))], Env$
$\texttt{in}\ t')$
$\longrightarrow (\texttt{letrec}$
$x = R^-[(\texttt{case}\ t\ (pat_1 \to (\texttt{letrec}\ y = s\ \texttt{in}\ t_1))$
$\ldots$
$(pat_N \to (\texttt{letrec}\ y = s\ \texttt{in}\ t_N)))],$
Env
$\texttt{in}\ t')$
if y does not occur free in R^-, Env, s, t' and t.

(ucpb-in) $(\texttt{letrec}\ x = s, Env\ \texttt{in}\ \ S_1[(\texttt{case}\ t\ \ldots (pat_i \to S_2[x]) \ldots)])$
$\longrightarrow (\texttt{letrec}\ x = s, Env\ \texttt{in}\ \ S_1[(\texttt{case}\ t\ \ldots (pat_i \to S_2[s]) \ldots)])$
if x does not occur free in Env, S_1, S_2, t and s.

(ucpb-e) $(\texttt{letrec}\ x = s, Env, y = S_1[(\texttt{case}\ t \ldots (pat_i \to S_2[x]) \ldots)]\ \ \texttt{in}\ t_1)$
$\longrightarrow (\texttt{letrec}\ x = s, Env, y = S_1[(\texttt{case}\ t \ldots (pat_i \to S_2[s]) \ldots)]\ \ \texttt{in}\ t_1)$
if x does not occur free in Env, S_1, S_2, t, t_1 and s.

Figure 6: Transformations for copying expressions

Proof. See [Sab03b, Theorem 3.75]. □

In the next section we will investigate a lot of program transformations, which are performed in the GHC. We have proven them to be correct by using the results of this section.

3 The relation between FUNDIO and Haskell

3.1 Our representation of the core language of the GHC

Definition 3.1. ($L_{GHCCore}$) *The language* $L_{GHCCore}$ *is defined in figure 7. We will also call this language* GHC core language. *Bold symbols are nonterminals, whose definition is given, italic symbols are other nonterminals; all other symbols are terminals. A valid expression (program) can be derived starting with nonterminal* **Expr** *(***Prog***).*

Prog	::=	$\mathbf{Binding}_1; \ldots; \mathbf{Binding}_n$	$n \geq 1$
Binding	::=	**Bind**	
	|	`rec` $\mathbf{Bind}_1; \ldots; \mathbf{Bind}_n$	
Bind	::=	*Var* `=` **Expr**	
Expr	::=	**Expr Expr**	(application)
	|	λ $Var_1 \ldots Var_n$ `->` **Expr**	(abstraction)
	|	`case` **Expr** `of` **Alts**	(case expression)
	|	`let` **Binding** `in` **Expr**	(local definition)
	|	*Var*	(variable)
	|	*Con*	(constructor)
	|	**Literal**	(unboxed object)
	|	*Prim*	(primitive operator)
Literal	::=	*Int* | *Char* | ...	
Alts	::=	$\mathbf{Calt}_1; \ldots; \mathbf{Calt}_n$; [**Default**]	$n \geq 0$
	|	$\mathbf{Lalt}_1; \ldots; \mathbf{Lalt}_n$; [**Default**]	$n \geq 0$
Calt	::=	*Con* $Var_1 \ldots Var_n$ `->` **Expr**	$n \geq 0$
Lalt	::=	**Literal** `->` **Expr**	
Default	::=	*Var* `->` **Expr**	

Figure 7: $L_{GHCCore}$ – The GHC core language

Additionally to the presented grammar the following conditions must hold:

- *A valid program has a top-level binding with left hand side* `main`*.*
- *Constructor applications or applications to primitive operators need not be saturated, but the number of arguments must not be greater then the arity and inside patterns only saturated constructor applications are allowed.*
- *The* `case` *alternatives are exhaustive insofar as for every constructor to which the scrutinee can be reduced a pattern is given.*
- *There is no* `case` *expression with alternatives for constructors from different types, except for a* `case` *expression, whose alternatives consist only of a* `default`*-alternative.*

We use the following conventions for the representation of terms on the GHC core language: Parenthesis are used to avoid ambiguities. The application is left-associative and binds stronger then every other operator. The body of an abstraction reaches as far as possible. We use arithmetic operators infix. If the meaning is clear, we omit semicolons between bindings and alternatives. We use the notation $f\ a_1\ \ldots\ a_n = e$ for functions, where the meaning is always $f = \lambda\ a_1 \ldots a_n$ `->` e. We say an expression is *atomic* if the expression is a literal or a variable.

The representation of $L_{GHCCore}$ is similar to [San95] and [PS94], but it has been adjusted to the actual core language of GHC, which has been derived from [Apt] and [PM02, page 400] and of course from the source code of the GHC[3]. We point out some differences between our representation of $L_{GHCCore}$ and the real core language, which is used in the compiler:

- The language inside the GHC is explicitly typed (by further language constructs). We ignore types whenever possible. Inside the syntax we have no types, but we assume that the set of constructors of $L_{GHCCore}$ is partitioned, where every partition relates to a type. For example, the constructors `True` and `False` build a partition of the former type `Bool`.
- In the GHC `case` expressions have a different representation of the following form:

  `case` **Expr** `of` Var **Alts**

  The additional variable Var is called the "`case`-binder", where the semantics is, that after evaluating the scrutinee the result is bound to Var. Accordingly, in reality the default-alternative does not introduce a fresh variable, it is represented as `DEFAULT ->` **Expr**, where `DEFAULT` is a constant, which can only occur as a pattern.
- The language inside the GHC has an additional construct **Note Expr** to mark expressions with some additional information.

[3]The core language is defined in the module `ghc/compiler/coreSyn/CoreSyn.lhs`. We refer to modules of the GHC with the whole directory path corresponding to the directory structure of the source distribution of GHC 5.04.3.

3.2 Translating the GHC core language to FUNDIO

3.2.1 The translation

We introduce the translation $[\![\cdot]\!]$, which translates (untyped) expressions of $L_{GHCCore}$ to L_{FUNDIO}.

Definition 3.2. (Translation $[\![\cdot]\!]$) *Let $e \in L_{GHCCore}$. Then $[\![e]\!] \in L_{FUNDIO}$ is the translated expression. Figure 8 presents most of the translation rules. We divide the steps of translating an expression with the symbol $\equiv$.*

The translation of an expression is done top-down step by step based on the term structure of the expression. The translation is meaningful, because the constructs like `case`, `letrec`, abstractions and applications are translated in the same constructs in L_{FUNDIO} whenever this is possible. We regard some special cases: In $L_{GHCCore}$ alternatives of `case` expressions do not have patterns for every constructor, but in L_{FUNDIO} this is necessary. Therefore, we add enough alternatives while translating where the right hand sides are all bot-terms. `case` expressions, which have a default alternative cannot be translated directly, because L_{FUNDIO} has no default construct. Therefore, we translate those expressions into `case` expressions with a single alternative for every constructor which is matched by the default alternative. Additionally we add a surrounding `letrec` construct, to share the evaluated value, as the default alternative does. FUNDIO does not provide something like unboxed values. But these primitive values are only a finite set of values. So we translate every of those values as a constant (the constants are added to the set of constructors $\mathcal{C}$ of the FUNDIO calculus).

Translation rules for primitive operators are missing, because every of those operators needs a more or less special treatment. We present the translation of those operators informally by translating some examples. Primitive operators without side-effects are translated into functions which test all possible combinations of inputs (this is a finite set) and return the corresponding constant. So these functions are strict in all of their arguments, where the strictness is generated by using additionally `case` expressions. For example, the primitive addition (`+#`) over two values of type `Int#` is translated as follows:

$$
\begin{array}{lll}
[\![\texttt{+\#}]\!] \equiv & (\lambda a_1.(\lambda a_2.(\texttt{case}\ a_1 & ([\![\texttt{-2147483648\#}]\!] \rightarrow\ \texttt{case}\ a_2 \ldots) \\
 & & ([\![\texttt{-2147483647\#}]\!] \rightarrow\ \texttt{case}\ a_2 \ldots) \\
 & & \ldots \\
 & & ([\![\texttt{1\#}]\!] \rightarrow\ \texttt{case}\ a_2 \ldots ([\![\texttt{1\#}]\!] \rightarrow [\![\texttt{2\#}]\!])([\![\texttt{2\#}]\!] \rightarrow [\![\texttt{3\#}]\!]) \ldots) \\
 & & \ldots \\
 & & ([\![\texttt{2147483647\#}]\!] \rightarrow\ \texttt{case}\ a_2 \ldots) \\
 & & (p_n \rightarrow \bot) \ldots (p_N \rightarrow \bot)\ )))
\end{array}
$$

Operators with side-effects are translated by using the `IO` construct of the FUNDIO calculus. We assume that `getChar` and `putChar` are primitive operators, and translate them as follows:

program: $[\![binding_1;\ldots;\text{main}=t;\ldots;binding_n]\!]$
$\equiv (\texttt{letrec}\ [\![binding_1]\!],\ldots,\text{main}=[\![t]\!],\ldots,\ [\![binding_n]\!]\ \texttt{in}\ \text{main})$

$[\![binding_1;\ldots;\texttt{rec}\ \{bind_{i,1};\ldots;\text{main}=t;\ldots;bind_{i,n_i}\};\ldots;binding_n]\!]$
$\equiv (\texttt{letrec}\ [\![binding_1]\!],\ldots,$
$[\![bind_{i,1}]\!],\ldots,\text{main}=[\![t]\!],\ldots,[\![bind_{i,n_i}]\!],$
$\ldots,[\![binding_n]\!]$
$\texttt{in}\ \text{main})$

bindings: $[\![x=t]\!]\ \equiv\ x=[\![t]\!]$

$[\![\texttt{rec}\ \{x_1=t_1;\ldots;x_n=t_n\}]\!]\ \equiv\ x_1=[\![t_1]\!],\ldots,x_n=[\![t_n]\!]$

application: $[\![t\ a]\!]\ \equiv\ ([\![t]\!]\ [\![a]\!])$

abstraction: $[\![\lambda\text{var}_1\ldots\text{var}_n\texttt{->}\ t]\!]\ \equiv\ (\lambda\text{var}_1.\ (\ldots(\lambda\text{var}_n.\ [\![t]\!])\ldots))$

let: $[\![\texttt{let}\ v=s\ \texttt{in}\ t]\!]\ \equiv\ (\texttt{letrec}\ v=[\![s]\!]\ \texttt{in}\ [\![t]\!])$

letrec: $[\![\texttt{letrec}\ v_1=s_1;\ldots;v_n=s_n\ \texttt{in}\ t]\!]$
$\equiv\ (\texttt{letrec}\ v_1=[\![s_1]\!],\ldots,v_n=[\![s_n]\!]\ \texttt{in}\ [\![t]\!])$

constructor: $[\![c]\!]\ \equiv\ (\lambda x_1.(\lambda x_2.\ldots(\lambda x_{ar(c)}.(c\ x_1\ldots x_{ar(c)}))\ldots))$

variable: $[\![x]\!]\ \equiv\ x$, if x is variable.

literal: $[\![\textit{unboxed value}]\!]\ \equiv\ c_i$,
where for every unboxed value a special constant c_i exists.

pattern: $[\![c\ a_1\ldots a_{ar(c)}]\!]\ \equiv\ (c\ a_1\ldots a_{ar(c)})$, if $c\ a_1\ldots a_{ar(c)}$ is a pattern.

case without a default alternative:
$[\![\texttt{case}\ t\ \texttt{of}\ pat_1\ \texttt{->}\ t_1;\ \ldots\ pat_n\ \texttt{->}\ t_n;]\!]$
$\equiv(\texttt{case}\ [\![t]\!]\ ([\![pat_1]\!]\to[\![t_1]\!])\ldots([\![pat_n]\!]\to[\![t_n]\!])\ (pat_{n+1}\to\bot)\ldots(pat_N\to\bot))$
where $\bot$ is a bot-term, $pat_{n+1},\ldots pat_N$ are patterns for the constructors of $\mathcal{C}$ which are not covered through the given patterns, i.e. if pat_i covers the constructor c_i, then $pat_i=c_i\ a_1\ \ldots\ a_{ar(c_i)}$, for $i=n+1,\ldots,N-1$ and $pat_N=\texttt{lambda}$.

case with alternatives including a default alternative:
$[\![\texttt{case}\ t\ \texttt{of}\ pat_1\ \texttt{->}\ t_1;\ldots;pat_n\ \texttt{->}\ t_n;x\ \texttt{->}\ s]\!]$
$\equiv(\texttt{letrec}\ y=[\![t]\!]\ \texttt{in}\ (\texttt{case}\ y\ ([\![pat_1]\!]\to[\![t_1]\!])\ldots([\![pat_n]\!]\to[\![t_n]\!])$
$([\![pat_{n+1}]\!]\to[\![s[y/x]]\!]))\ldots([\![pat_m]\!]\to[\![s[y/x]]\!]))$
$(pat_{m+1}\to\bot)\ \ldots(pat_N\to\bot))$
if pat_i, $i=1,\ldots,n$ are patterns of a type with $m\geq n$ constructors. $pat_{n+1},\ldots pat_m$ are the missing patterns for constructors of this type. $pat_{m+1},\ldots pat_N$ cover the remaining constructors in L_{FUNDIO}. y is a fresh variable.

case only with a default alternative:
$[\![\texttt{case}\ t\ \texttt{of}\ x\ \texttt{->}\ s]\!]$
$\equiv(\texttt{letrec}\ y=[\![t]\!]\ \texttt{in}\ (\texttt{case}\ y\ (pat_1\to[\![s[y/x]]\!])\ldots(pat_N\to[\![s[y/x]]\!])))$
where y is a fresh variable.

Figure 8: Translation from $L_{GHCCore}$ to L_{FUNDIO}

$$
\begin{aligned}
[\![\texttt{getChar}]\!] &\equiv ([\![IO]\!]\ (\lambda w.(\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to (w, p_1)) \ldots (p_n \to (w, p_n)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))) \\
[\![\texttt{putChar}]\!] &\equiv (\lambda x.([\![IO]\!]\ (\lambda w.(\texttt{case}\ x \\
&\qquad\qquad (p_1 \to (\texttt{case}\ (\texttt{IO}\ x)\ (p_1 \to (w, [\![\texttt{()}]\!])) \ldots (p_N \to (w, [\![\texttt{()}]\!])))) \\
&\qquad\qquad \ldots \\
&\qquad\qquad (p_n \to (\texttt{case}\ (\texttt{IO}\ x)\ (p_1 \to (w, [\![\texttt{()}]\!])) \ldots (p_N \to (w, [\![\texttt{()}]\!])))) \\
&\qquad\qquad (p_{n+1} \to \bot) \ldots (p_N \to \bot)))))
\end{aligned}
$$

where $p_1, \ldots, p_n$, are patterns for constructors of the charset, which is a subset of $\mathcal{C}$, $p_{n+1}, \ldots p_N$ are patterns for the remaining constructors of $\mathcal{C}$, $\mathcal{B}$ is a special "blank symbol" of $\mathcal{C}$, and $[\![IO]\!]$ ($[\![\texttt{()}]\!]$) is the translation of the constructor IO (`()`) of the GHC core language.

The translation of `getChar` can be derived as follows: Because `getChar` is an IO-action, the returned expression is a – boxed by the IO constructor – function which receives a state of the world and returns a pair consisting of the new state and a character. The `case` construct ensures, that the `IO` expression is evaluated before the new state is returned and that only characters are accepted as result.

3.2.2 Examples

We present, how the function `unsafePerformIO` is translated into FUNDIO and illustrate the coherence between `unsafePerformIO` and the nondeterministic `IO` of the FUNDIO calculus.

A slightly simplified definition of `unsafePerformIO` in Haskell is:

```
unsafePerformIO (IO m) = case m realWorld# of (s, r) -> r
```

This expression can be presented in $L_{GHCCore}$ in the following way:

$$
\begin{aligned}
\texttt{unsafePerformIO} = \lambda i\ \texttt{->}\ \texttt{case}\ i\ \texttt{of} \\
(IO\ m)\ \texttt{->}\ \texttt{case}\ m\ \text{realWorld\#}\ \texttt{of} \\
(s, r)\ \texttt{->}\ r
\end{aligned}
$$

Example 3.3. *By translating and simplifying by program transformations we have shown in [Sab03b]:*

$$
\begin{aligned}
& [\![\texttt{unsafePerformIO getChar}]\!] \\
\sim_c\ & (\texttt{case}\ (\texttt{IO}\ \mathcal{B})\ (p_1 \to p_1) \ldots (p_n \to p_n)\ (p_{n+1} \to \bot) \ldots (p_N \to \bot))
\end{aligned}
$$

Here $p_1, \ldots, p_n$ are patterns for the elements of the charset. The translation is similar to the nondeterministic `IO` *construct of FUNDIO, where the additionally* `case` *expression arises from the fact, that* `getChar` *returns only characters and no other constants.*

Example 3.4. *Also in [Sab03b] we have shown:*

$$\begin{array}{rl} & [\![\lambda c \texttt{ -> unsafePerformIO } (\texttt{putChar } c)]\!] \\ \sim_c & (\lambda c.(\texttt{case } c\ (p_1 \to (\texttt{case } (\texttt{IO } c)\ (p_1 \to ([\![\texttt{()}]\!]))\ldots\ (p_N \to ([\![\texttt{()}]\!])))) \\ & \qquad\qquad \ldots \\ & \qquad\qquad (p_n \to (\texttt{case } (\texttt{IO } c)\ (p_1 \to ([\![\texttt{()}]\!]))\ldots\ (p_N \to ([\![\texttt{()}]\!])))) \\ & \qquad\qquad (p_{n+1} \to \bot)\ldots(p_N \to \bot))) \end{array}$$

The expression is similar to $(\lambda c.(\texttt{IO } c))$, *where the additional* `case` *expressions ensure that only characters are printed, as well as that the input-value is ignored and the translation of* `()` *is always returned.*

The translation $[\![\cdot]\!]$ transforms constructors with positive arity into abstractions. Accordingly, constructor applications are translated into applications to abstractions. We now show, that saturated constructor applications can be translated directly into L_{FUNDIO}.

Example 3.5. *Let* $c\ a_1 \ldots a_n \in L_{GHCCore}$ *be a saturated constructor application, then the translated expression in* L_{FUNDIO} *is contextually equivalent to the constructor application* $([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!])$:

$$\begin{array}{rl} & [\![c\ a_1 \ldots a_n]\!] \\ \equiv & (\ldots((\lambda x_1.(\ldots(\lambda x_n.([\![c]\!]\ x_1 \ldots x_n))\ldots))\ [\![a_1]\!])\ldots[\![a_n]\!]) \\ \xrightarrow{lbeta} & (\texttt{letrec } x_1 = [\![a_1]\!] \texttt{ in } (\ldots((\lambda x_2.(\ldots(\lambda x_n.([\![c]\!]\ x_1 \ldots x_n))\ldots))\ [\![a_2]\!])\ldots[\![a_n]\!])) \\ \xrightarrow{(lll)^*} & (\texttt{letrec } x_1 = [\![a_1]\!], \ldots x_n = [\![a_n]\!] \texttt{ in } ([\![c]\!]\ x_1 \ldots x_n)) \\ \xrightarrow{(ucp)^*} & (\texttt{letrec } \{\} \texttt{ in } ([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!])) \\ \xrightarrow{gc} & ([\![c]\!]\ [\![a_1]\!] \ldots [\![a_n]\!]) \end{array}$$

3.2.3 Correctness of program transformations on the GHC core language

We define the *correctness of a program transformation* in $L_{GHCCore}$ by firstly translating the transformation into L_{FUNDIO} and secondly using the contextual equivalence of the FUNDIO calculus.

Definition 3.6. ($[\![\cdot]\!]$-correctness) *Let P be a program transformation on expressions $s, t \in L_{GHCCore}$. We say P is* $[\![\cdot]\!]$-correct *if the following holds:* $s\ P\ t \implies [\![s]\!] \sim_c [\![t]\!]$

With regard to that correctness we will investigate a lot of program transformations, which are performed by the GHC.

3.3 Classification of the transformations on GHC core

We divide the transformations on the GHC core language as in [PS94, San95, PS98] into two classes: The first class consists of *local transformations* which transform small subexpressions. The power of these transformations arises from performing them together and more then once iteratively. The local transformations are performed by the so-called “simplifier”. The *global transformations* like strictness analysis or “common subexpression elimination” form the second class of transformations. Nearly each of these transformations is implemented as one compiler pass and can be turned on or off separately. After performing such a compiler pass the simplifier is called to clean up the code. Therefore, it is important that only correct local transformations are performed, so we will analyse them in detail in the next section. The global transformations are not treated in detail, but in Section 3.5 we give a brief summary of them with some comments.

3.4 Local transformations

In this section we investigate the local transformations, which are performed in the GHC. The presented transformations and their effects are described in detail in [PS94, San95], but the underlying core language in this papers differs from the one currently in use and from $L_{GHCCore}$. Therefore, we have adapted the transformations to the current implementation. We denote a transformation with the name *rule*, which transforms expressions of form l into expressions of form r as

$$l \overset{(rule)}{\texttt{===>}} r.$$

In [Sab03b] we have analyzed every of the presented transformations, where we have shown the $[\![\cdot]\!]$-correctness of a transformation by transforming $[\![l]\!]$ into $[\![r]\!]$ by using the correct program transformations of [SS03] and Theorem 2.21. In this paper we do not present the proofs again. Instead, we present our results and sketch some of the proofs. If a transformation is not correct, we will give counter-examples. Analogously to “contexts” for the FUNDIO calculus we use contexts in $L_{GHCCore}$ without giving an explicit definition here.

3.4.1 Variants of beta reduction

Atomic beta-reduction

$$(\lambda x \texttt{ -> } e)\ arg \overset{(\beta\text{-atom})}{\texttt{===>}} e[arg/x], \qquad \text{if } arg \text{ is atomic.}$$

Beta with sharing

$$(\lambda x \texttt{ -> } e)\ arg \overset{(\beta)}{\texttt{===>}} \texttt{let } x = arg \texttt{ in } e$$

Figure 9: Variants of beta-reduction

Figure 9 shows two variants of beta reduction. (β-atom) is ordinary beta reduction for atomic arguments, (β) is a variant of beta reduction, which shares the argument. (β-atom) and (β) are $[\![\cdot]\!]$-correct program transformations. The proofs are easy, because (β) is similar to the (lbeta) rule of FUNDIO and the $[\![\cdot]\!]$-correctness of (β-atom) can be proven by using the (beta-var) rule if the argument is a variable. If the argument is a literal, the translation of the argument is a constant. Then the $[\![\cdot]\!]$-correctness can be shown, by using the rules (lbeta), (cpcx) and (gc).

3.4.2 Transformations on let(rec)-expressions

Figure 10 shows some transformations on `let(rec)` expressions. *Floating let out of let* and *floating let out of a case scrutinee* are $[\![\cdot]\!]$-correct, where the proofs are trivial because of the similar (llet) and (lcase) rules of FUNDIO. By using the (gc) rule of the FUNDIO calculus the *dead code removal* transformations, which are used to eliminate unused bindings, can be proven to be $[\![\cdot]\!]$-correct. The transformation for general *inlining* is not $[\![\cdot]\!]$-correct, which is shown by the following counter-example.

Example 3.7. *Let $s \in L_{GHCCore}$ be the following expression:*

$$s := \texttt{let } x = (\texttt{unsafePerformIO getChar}) \texttt{ in case } x \texttt{ of 'd' -> (case } x \texttt{ of 'd' -> 'd')}$$

We can obtain the following expression t by one application of the (inl) transformation.

$$\begin{aligned} t := \; & \texttt{let } x = (\texttt{unsafePerformIO getChar}) \texttt{ in} \\ & \quad \texttt{case } (\texttt{unsafePerformIO getChar}) \texttt{ of 'd' -> (case } x \texttt{ of 'd' -> 'd')} \end{aligned}$$

Let $P = \{(\mathcal{B}, \texttt{'d'})\}$, then $[\![s]\!]\Downarrow(P)$, but $\neg([\![t]\!]\Downarrow(P))$, i.e. $[\![s]\!] \not\sim_c [\![t]\!]$.

Figure 10 shows some special forms of inlining, which were developed after browsing the source code of GHC. *Unique inlining* is similar to the (ucp) rule of the FUNDIO calculus and hence (uinl) can be shown to be $[\![\cdot]\!]$-correct by using this rule. Similar to the (ucpb-in) rule of FUNDIO we have defined the (bruinl) transformation. By using the (ucpb) rule, we have shown in [Sab03b], that (bruinl) is a $[\![\cdot]\!]$-correct program transformation. For understanding *cheap inlining* we firstly define the language $CHEAP$.

Definition 3.8. ($CHEAP$) *Let $CHEAP$ be the following set of expressions of $L_{GHCCore}$:*

$x \in CHEAP$ iff.

- *x is a literal,*
- *x is a variable,*
- *x is an abstraction,*
- *x is a primitive operator with arity > 0, or*
- *x is a constructor application $c_i\ a_1 \ldots a_n$, $n \leq ar(c_i)$ and $a_j \in CHEAP$ for $j = 1, \ldots, n$*

Floating let out of let

Rule for let:

$$\begin{array}{l}\texttt{let}\ x = (\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in}\ B_1)\\ \texttt{in}\ B_2\end{array} \overset{\text{(flool-let)}}{\texttt{===>}} \begin{array}{l}\texttt{let(rec)}\ \mathit{Bind}\\ \texttt{in}\ (\texttt{let}\ x = B_1\ \texttt{in}\ B_2)\end{array}$$

Rule for letrec:

$$\begin{array}{l}\texttt{letrec}\ x = (\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in}\ B_1)\\ \texttt{in}\ B_2\end{array} \overset{\text{(flool-letrec)}}{\texttt{===>}} \begin{array}{l}\texttt{letrec}\ \mathit{Bind}; x = B_1\\ \texttt{in}\ B_2\end{array}$$

Floating let out of a case scrutinee

$$\texttt{case}\ (\texttt{let(rec)}\ \mathit{Bind}\ \texttt{in}\ E)\ \texttt{of}\ \mathit{Alts} \overset{\text{(flooacs)}}{\texttt{===>}} \begin{array}{l}\texttt{let(rec)}\ \mathit{Bind}\\ \texttt{in case}\ E\ \texttt{of}\ \mathit{Alts}\end{array}$$

Dead code removal

Rule for let:

$$\texttt{let}\ x = E\ \texttt{in}\ B \overset{\text{(dcr-let)}}{\texttt{===>}} B, \qquad \text{if } x \text{ has no free occurrence in } B.$$

Rule for letrec:

$$\texttt{rec}\ \mathit{bindings}\ \texttt{in}\ B \overset{\text{(dcr-letrec)}}{\texttt{===>}} B, \qquad \text{if none of the } \mathit{bindings} \text{ is used in } B$$

Inlining

$$\texttt{let(rec)}\ x = e\ \texttt{in}\ C[x] \overset{\text{(inl)}}{\texttt{===>}} \texttt{let(rec)}\ x = e\ \texttt{in}\ C[e]$$

Unique inlining

$$\texttt{let(rec)}\ x = e\ \texttt{in}\ C[x] \overset{\text{(uinl)}}{\texttt{===>}} C[e]$$

if x occurs free exactly once in $C[x]$, but not in a body of an abstraction, and x does not occur free in e.

Branch unique inlining

$$\begin{array}{l}\texttt{let(rec)}\ x = e\ \texttt{in}\\ \quad C[\texttt{case}\ e_1\ \texttt{of}\\ \qquad\quad P_1\ \texttt{->}\ B_1\\ \qquad\quad \ldots\\ \qquad\quad P_i\ \texttt{->}\ C'[x]\\ \qquad\quad \ldots\\ \qquad\quad P_n\ \texttt{->}\ B_n]\end{array} \overset{\text{(bruinl)}}{\texttt{===>}} \begin{array}{l}\texttt{let(rec)}\ x = e\ \texttt{in}\\ \quad C[\texttt{case}\ e_1\ \texttt{of}\\ \qquad\quad P_1\ \texttt{->}\ B_1\\ \qquad\quad \ldots\\ \qquad\quad P_i\ \texttt{->}\ C'[e]\\ \qquad\quad \ldots\\ \qquad\quad P_n\ \texttt{->}\ B_n]\end{array}$$

if x occurs only in $B_1, \ldots, B_n$ and occurs free exactly once in $C'[x]$, where the occurrence in $C[C'[x]]$ is not in a body of an abstraction.

Cheap inlining

$$\texttt{let(rec)}\ x = e\ \texttt{in}\ C[x] \overset{\text{(cheapinl)}}{\texttt{===>}} \texttt{let(rec)}\ x = e\ \texttt{in}\ C[e], \qquad \text{if } e \in \mathit{CHEAP}.$$

Figure 10: Transformations on `let(rec)` expressions

The definition of $CHEAP$ was inspired from GHC's "cheap" expressions[4], but in the GHC more expressions are allowed to be "cheap", so our set is smaller than that used in the GHC. Note that the following holds: $s \in CHEAP \implies [\![s]\!] \in L_{cheap}$. (cheapinl) is $[\![\cdot]\!]$-correct which can be proven by using the (cheapcp) rule of FUNDIO.

3.4.3 Transformations on case-expressions

The transformations on `case`-expressions are defined in the figures 11 and 12.

The *case of known constructor* transformation described in [San95, PS94] does no sharing, but the current implementation[5] and also the defined (cokc) rule respects sharing. In [Sab03b] we have shown, that (cokc) is $[\![\cdot]\!]$-correct. Analogous variants, where the constructor application is bound to a variable and the arguments are atomic are defined as (cokc-l) and (cokc-c). The $[\![\cdot]\!]$-correctness of (cokc-l) can easily be shown, because the constructor application with atomic arguments can be copied in FUNDIO with the (cpcheap) rule. After that the proof of the (cokc) can be used. The (cokc-c) is $[\![\cdot]\!]$-correct, because by using the (ccpcx) rule of FUNDIO the constructor application $(c\ x_1 \dots x_n)$ can be copied into the alternative and then the proof of the (cokc) transformation can be used for the inner `case` expression. Finally the arisen `letrec` expression can be eliminated by doing some (cpcheap) and a (dcr-letrec) transformation.

The (cokc-default) transformation is a variant of the case, that no pattern of an alternative matches, but a default alternative is given. In [Sab03b] we have shown, that (cokc-default) is a $[\![\cdot]\!]$-correct program transformation.

By using the (cpx) rule of FUNDIO, we have shown that *default binding elimination* is $[\![\cdot]\!]$-correct.

Dead alternative elimination is used to eliminate unreachable `case` alternatives. In [Sab03b] we have shown, that (dae) is a $[\![\cdot]\!]$-correct program transformation, by using the (crpl) rule of FUNDIO.

The function `error` has the semantic value $\perp$. So, the translation of this function is a bot-term. By using Lemma 2.15 it is easy to show that the *case of error*-transformation is $[\![\cdot]\!]$-correct.

Floating case out of case has been shown to be $[\![\cdot]\!]$-correct in [Sab03b] by using the (ccase) and (ccase-in) rule of FUNDIO. The (fcooc) transformation increases the size of the code (the m alternatives exist n times after performing the transformation). In the GHC this transformation is performed in another way by using so-called "join points", i.e. the right hand sides of the alternatives are shared as follows: Let $Q_i = c_i\ y_{i,1} \dots y_{i,n_i}$ for $i = 1, \dots, m$, then the right hand side of the transformation has the form:

[4] In the module `ghc/compiler/coreSyn/CoreUtils.lhs` the predicate `exprIsCheap` is defined.
[5] In module `ghc/compiler/simplCore/Simplify.lhs` the function `knownCon` is defined.

Case of known constructor

General rule:

$$\begin{array}{l}\texttt{case}\ (c\ a_1 \ldots a_n)\ \texttt{of} \\ \quad \ldots \\ \quad c\ b_1 \ldots b_n\ \texttt{->}\ e \\ \quad \ldots\end{array} \quad \overset{\text{(cokc)}}{\texttt{===>}} \quad \begin{array}{l}\texttt{letrec}\ b_1 = a_1; \ldots ; b_n = a_n \\ \texttt{in}\ e\end{array}$$

Rule for a let-bound scrutinee:

$$\begin{array}{l}\texttt{let(rec)}\ x = c\ a_1 \ldots a_n\ \texttt{in} \\ \ \texttt{case}\ x\ \texttt{of} \\ \quad c\ b_1 \ldots b_n\ \texttt{->}\ \ e \\ \quad \ldots\end{array} \quad \overset{\text{(cokc-l)}}{\texttt{===>}} \quad \begin{array}{l}\texttt{let(rec)}\ x = c\ a_1 \ldots a_n \\ \texttt{in letrec}\ b_1 = a_1, \ldots , b_n = a_n \\ \quad \texttt{in}\ e\end{array}$$

Rule for a case-bound scrutinee

$$\begin{array}{l}\texttt{case}\ x\ \texttt{of} \\ \ c\ x_1 \ldots x_n\ \texttt{->}\ \texttt{case}\ x\ \texttt{of} \\ \qquad\qquad\qquad\quad c\ y_1 \ldots y_n\ \texttt{->}\ e \\ \qquad\qquad\qquad\quad \ldots \\ \ \ldots\end{array} \quad \overset{\text{(cokc-c)}}{\texttt{===>}} \quad \begin{array}{l}\texttt{case}\ x\ \texttt{of} \\ \ c\ x_1 \ldots x_n\ \texttt{->}\ e[x_i/y_i]_{i=1}^n \\ \ \ldots\end{array}$$

Case of known constructor with a matching default alternative

$$\begin{array}{l}\texttt{case}\ (c\ a_1 \ldots a_n)\ \texttt{of} \\ \ \ldots \\ \ y\ \texttt{->}\ E\end{array} \quad \overset{\text{(cokc-default)}}{\texttt{===>}} \quad \begin{array}{l}\texttt{let}\ y = (c\ a_1 \ldots a_n) \\ \texttt{in}\ E\end{array} ,$$

if only the default alternative matches.

Default binding elimination

$$\texttt{case}\ v_1\ \texttt{of}\ v_2\ \texttt{->}\ e \overset{\text{(dbe)}}{\texttt{===>}} \texttt{case}\ v_1\ \texttt{of}\ v_2\ \texttt{->}\ e[v_1/v_2], \qquad \text{where } v_1 \text{ and } v_2 \text{ are variables.}$$

Dead alternative elimination

$$\begin{array}{l}\texttt{case}\ x\ \texttt{of} \\ \quad (c_1\ a_{1,1} \ldots a_{1,ar(c_1)})\ \texttt{->}\ E_1; \\ \quad \ldots; \\ \quad (c_k\ a_{k,1} \ldots a_{k,ar(c_k)})\ \texttt{->}\ E_k; \\ \quad \ldots; \\ \quad (c_n\ a_{n,1} \ldots a_{n,ar(c_n)})\ \texttt{->}\ E_n;\end{array} \quad \overset{\text{(dae)}}{\texttt{===>}} \quad \begin{array}{l}\texttt{case}\ x\ \texttt{of} \\ \quad (c_1\ a_{1,1} \ldots a_{1,ar(c_1)})\ \texttt{->}\ E_1; \\ \quad \ldots; \\ \quad (c_{k-1}\ a_{k-1,1} \ldots a_{k-1,ar(c_{k-1})})\ \texttt{->}\ E_{k-1}; \\ \quad (c_{k+1}\ a_{k+1,1} \ldots a_{k+1,ar(c_{k+1})})\ \texttt{->}\ E_{k+1}; \\ \quad \ldots; \\ \quad (c_n\ a_{n,1} \ldots a_{n,ar(c_n)})\ \texttt{->}\ E_n;\end{array} ,$$

if x is not of constructor c_k.

Case of error

$$\texttt{case}\ (\texttt{error}\ E)\ \texttt{of}\ Alts \overset{\text{(coe)}}{\texttt{===>}} \texttt{error}\ E$$

Figure 11: Transformations on `case` expressions

Floating case out of case

$$\texttt{case}\ \left(\begin{array}{l}\texttt{case}\ E\ \texttt{of}\\ \quad P_1\ \texttt{->}\ R_1\\ \quad \ldots;\\ \quad P_n\ \texttt{->}\ R_n\end{array}\right)\ \texttt{of} \qquad \overset{\text{(fcooc)}}{\texttt{===>}} \qquad \begin{array}{l}\texttt{case}\ E\ \texttt{of}\\ \quad P_1\ \texttt{->}\ \texttt{case}\ R_1\ \texttt{of}\\ \qquad\qquad Q_1\ \texttt{->}\ S_1\\ \qquad\qquad \ldots\\ \qquad\qquad Q_m\ \texttt{->}\ S_m\\ \quad \ldots\\ \quad P_n\ \texttt{->}\ \texttt{case}\ R_n\ \texttt{of}\\ \qquad\qquad Q_1\ \texttt{->}\ S_1\\ \qquad\qquad \ldots\\ \qquad\qquad Q_m\ \texttt{->}\ S_m\end{array}$$

$$\begin{array}{l}\quad Q_1\ \texttt{->}\ S_1\\ \quad \ldots\\ \quad Q_m\ \texttt{->}\ S_m\end{array}$$

Case merging

$$\begin{array}{l}\texttt{case}\ x\ \texttt{of}\\ \ c_1\ a_{1,1}\ldots a_{1,ar(c_1)}\ \texttt{->}\ t_1\\ \ \ldots\\ \ c_k\ a_{k,1}\ldots a_{k,ar(c_k)}\ \texttt{->}\ t_k\\ \ y\ \texttt{->}\\ \quad \texttt{case}\ x\ \texttt{of}\\ \quad\ c_{k+1}\ b_{k,1}\ldots b_{k,ar(c_{k+1})}\ \texttt{->}\ t_{k+1}\\ \quad\ \ldots\\ \quad\ c_m\ b_{m,1}\ldots b_{m,ar(c_m)}\ \texttt{->}\ t_m\end{array} \qquad \overset{\text{(cm)}}{\texttt{===>}} \qquad \begin{array}{l}\texttt{case}\ x\ \texttt{of}\\ \ c_1\ a_{1,1}\ldots a_{1,ar(c_1)}\ \texttt{->}\ t_1\\ \ c_k\ a_{k,1}\ldots a_{k,ar(c_k)}\ \texttt{->}\ t_k\\ \ c_{k+1}\ b_{k+1,1}\ldots b_{k+1,ar(c_{k+1})}\ \texttt{->}\ t_{k+1}[x/y]\\ \ c_m\ b_{m,1}\ldots b_{m,ar(c_m)}\ \texttt{->}\ t_m[x/y]\end{array}$$

where x is a variable.

Alternative merging

$$\begin{array}{l}\texttt{case}\ e\ \texttt{of}\\ \quad c_1\ a_{1,1}\ldots a_{1,m_1}\ \texttt{->}\ E_1;\\ \quad \ldots\\ \quad c_i\ a_{i,1}\ldots a_{i,m_i}\ \texttt{->}\ E;\\ \quad \ldots\\ \quad c_j\ a_{j,1}\ldots a_{j,m_j}\ \texttt{->}\ E;\\ \quad c_{j+1}\ a_{j+1,1}\ldots a_{j+1,m_{j+1}}\ \texttt{->}\ E_{j+1};\\ \quad \ldots\\ \quad c_n\ a_{n,1}\ldots a_{n,m_n}\ \texttt{->}\ E_n\end{array} \qquad \overset{\text{(am)}}{\texttt{===>}} \qquad \begin{array}{l}\texttt{case}\ e\ \texttt{of}\\ \quad c_1\ a_{1,1}\ldots a_{1,m_1}\ \texttt{->}\ E_1;\\ \quad \ldots\\ \quad c_{i-1}\ a_{i-1,1}\ldots a_{i-1,m_{i-1}}\ \texttt{->}\ E_{i-1};\\ \quad c_{j+1}\ a_{j+1,1}\ldots a_{j+1,m_{j+1}}\ \texttt{->}\ E_{j+1};\\ \quad \ldots\\ \quad c_n\ a_{n,1}\ldots a_{n,m_n}\ \texttt{->}\ E_n;\\ \quad y\ \texttt{->}\ E\end{array}$$

if for $k = i, \ldots, j$: $a_{k,1}\ldots a_{k,m_k}$ do not occur free in E.

Case identity

$$\texttt{case}\ e\ \texttt{of}\ \{P_1\ \texttt{->}\ P_1; \ldots; P_n\ \texttt{->}\ P_n\}\ \overset{\text{(ci)}}{\texttt{===>}}\ e$$

Case elimination

$$\texttt{case}\ e\ \texttt{of}\ y\ \texttt{->}\ E\ \overset{\text{(ce)}}{\texttt{===>}}\ \texttt{let}\ y = e\ \texttt{in}\ E, \qquad \text{if } e \neq \bot.$$

Figure 12: Transformations on `case` expressions (contd.)

```
letrec  s_1 = λy_{1,1} ... y_{1,n_1} -> S_1
        ...
        s_m = λy_{m,1} ... y_{m,n_m} -> S_m
in
 case E of
   P_1 -> case R_1 of
            c_1 y_{1,1} ... y_{1,n_1} -> s_1 y_{1,1} ... y_{1,n_1}
            ...
            c_m y_{m,1} ... y_{m,n_m} -> s_m y_{m,1} ... y_{m,n_m}
   ...
   P_n -> case R_n of
            c_1 y_{1,1} ... y_{1,n_1} -> s_1 y_{1,1} ... y_{1,n_1}
            ...
            c_m y_{m,1} ... y_{m,n_m} -> s_m y_{m,1} ... y_{m,n_m}
```

The use of join points is $[\![\cdot]\!]$-correct, because the s_i can be copied into the right hand sides of the alternatives, using the (bruinl) transformation. After that the bindings can be eliminated with (dcr-letrec). Finally in every alternative the (β-atom) transformation can be applied. The resulting expression corresponds to the right expression of the (fcooc) transformation.

Case merging "merges" the alternatives of nested `case` expressions. In [Sab03b] we have shown that (cm) is a $[\![\cdot]\!]$-correct program transformation.

In module `ghc/compiler/simplCore/SimplUtils.lhs` of the GHC *alternative merging* is performed by the function `mkAlts`, which unions `case` alternatives with identical right hand sides. Note that the `case`-alternatives on the left hand side of the rule need not contain a single alternative for every constructor, but the `case`-expression must be exhaustive as mentioned in Definition 3.1. Therefore, we have shown in [Sab03b] the $[\![\cdot]\!]$-correctness of (am) by using the (crpl) rule of FUNDIO.

The *case identity* transformation is performed in the module `ghc/compiler/simplCore/SimplUtils.lhs` by the function `mkCase1`. In [Sab03b] we have shown, that (ci) is $[\![\cdot]\!]$-correct, if the (streval) rule (defined in Definition 2.20) is a correct program transformation. Presumably, the $[\![\cdot]\!]$-correctness can be shown, without using the (streval) rule, by defining a similar rule for the FUNDIO-calculus and using the technique of complete sets of commuting and forking diagrams.

Case elimination is defined as used in the GHC[6]. Our definition differs from [PS94, San95], because we respect sharing. (ce) in general is not a $[\![\cdot]\!]$-correct program transformation, which is shown by the following counter-example:

Example 3.9. *Let s and t be the following expressions with $s \xrightarrow{(ce)} t$, where c is a*

[6]It is performed by the function `mkCase` in the module `ghc/compiler/simplCore/SimplUtils.lhs`.

constant:

$$s \;=\; \texttt{case}\ (\texttt{unsafePerformIO getChar})\ \texttt{of}\ y\ \texttt{->}\ c$$
$$t \;=\; \texttt{let}\ y = (\texttt{unsafePerformIO getChar})\ \texttt{in}\ c$$

Let $P = \emptyset$*, then* $\neg([\![s]\!]\Downarrow(P))$*, but* $[\![t]\!]\Downarrow(P)$*, i.e.* $[\![s]\!] \not\sim_c [\![t]\!]$

In [Sab03b] we have shown, that (ce) is $[\![\cdot]\!]$-correct, if e is an abstraction, a primitive operator (with positive arity), a literal or a (perhaps unsaturated) constructor application.

3.4.4 Transformations on let(rec)- and case-expressions

Figure 13 defines some transformations which all have a variant of `let(rec)` expressions and a variant of `case` expressions. The `let`-rule of *floating applications inwards* can be

Floating applications inwards

Rule for `let`:

$$(\texttt{let(rec)}\ Bind\ \texttt{in}\ E)\ arg \overset{\text{(fai-let)}}{\texttt{===>}} \texttt{let(rec)}\ Bind\ \texttt{in}\ (E\ arg)$$

Rule for `case`:

$$\begin{pmatrix} \texttt{case}\ E\ \texttt{of} \\ \quad P_1 \rightarrow E_1; \\ \quad \dots \\ \quad P_n \rightarrow E_n \end{pmatrix} arg \overset{\text{(fai-case)}}{\texttt{===>}} \begin{array}{l} \texttt{case}\ E\ \texttt{of} \\ \quad P_1 \rightarrow E_1\ arg; \\ \quad \dots \\ \quad P_n \rightarrow E_n\ arg \end{array}$$

Constructor reuse

Rule for `let`:

$$\begin{array}{l} \texttt{let}\ x = c\ a_1 \dots a_n \\ \texttt{in}\ C[c\ a_1 \dots a_n] \end{array} \overset{\text{(cr-let)}}{\texttt{===>}} \begin{array}{l} \texttt{let}\ x = c\ a_1 \dots a_n \\ \texttt{in}\ C[x] \end{array}, \quad \text{if the } a_i \text{ are atomic.}$$

Rule for `case`:

$$\begin{array}{l} \texttt{case}\ x\ \texttt{of} \\ \quad \dots \\ \quad c\ a_1 \dots a_n\ \texttt{->}\ \ C[c\ a_1 \dots\ a_n] \\ \quad \dots \end{array} \overset{\text{(cr-case)}}{\texttt{===>}} \begin{array}{l} \texttt{case}\ x\ \texttt{of} \\ \quad \dots \\ \quad c\ a_1 \dots a_n\ \texttt{->}\ C[x] \\ \quad \dots \end{array}, \quad \text{where } x \text{ is variable.}$$

Figure 13: Transformations on `let(rec)` and `case` expressions

shown to be $[\![\cdot]\!]$-correct by using the (lapp) rule of the FUNDIO calculus. The $[\![\cdot]\!]$-correctness of (fai-case) can be shown by using the (capp) rule of FUNDIO.

The rules for *constructor reuse* differ from those defined in [PS94, San95], because we added to the `let`-rule the condition, that the arguments of the constructor application are atomic. In [PS94, San95] this was not necessary, because of their core language. The condition holds also in the current implementation, because GHC allows only such `let`-bound constructor applications. This is mentioned in [PM02, page 399] and documented in the source code of the module `ghc/compiler/coreSyn/CoreSyn.lhs`. A constructor application with non-atomic arguments can be transformed into the demanded form by using the (uinl) transformation several times. For the FUNDIO calculus this procedure is described with the similar (ucp) rule in [SS03]. In [Sab03b] we have shown that (cr-case) and (cr-let) are $[\![\cdot]\!]$-correct.

3.4.5 Strictness-based transformations

The transformations shown in figure 14 need strictness information.

Let to case

$$\texttt{let}\ v = E_1\ \texttt{in}\ E_2 \overset{\text{(ltc)}}{\texttt{===>}} \texttt{case}\ E_1\ \texttt{of}\ v\ \texttt{->}\ E_2$$

if v has a constructor type, E_2 is strict in v and E_1 is not a WHNF.

Unboxing let to case

$$\texttt{let}\ v = E_1\ \texttt{in}\ E_2 \overset{\text{(ultc)}}{\texttt{===>}} \begin{array}{l}\texttt{case}\ E_1\ \texttt{of}\\ \quad c\ a_1 \ldots a_n\ \texttt{->}\ \texttt{let}\ v = c\ a_1 \ldots a_n\ \texttt{in}\ E_2\end{array}$$

if v has constructor type, which consists only of exactly one constructor c and E_2 is strict in v.

Floating case out of let

$$\begin{array}{l}\texttt{let}\ v = \texttt{case}\ E_1\ \texttt{of}\\ \qquad\qquad c_1\ a_{1,1} \ldots a_{1,m_1}\ \texttt{->}\ t_1;\\ \qquad\qquad \ldots\\ \qquad\qquad c_n\ a_{n,1} \ldots a_{n,m_n}\ \texttt{->}\ t_n\\ \texttt{in}\ E_3\end{array} \overset{\text{(fcool)}}{\texttt{===>}} \begin{array}{l}\texttt{case}\ E_1\ \texttt{of}\\ \quad c_1\ a_{1,1} \ldots a_{1,m_1}\ \texttt{->}\ \texttt{let}\ v = t_1\ \texttt{in}\ E_3;\\ \quad \ldots\\ \quad c_n\ a_{n,1} \ldots a_{n,m_n}\ \texttt{->}\ \texttt{let}\ v = t_n\ \texttt{in}\ E_3\end{array}$$

if E_3 is strict in v and v is not free in E_1.

Figure 14: Strictness-based transformations

The *let to case* transformation uses strictness information to evaluate a `let` bound expression earlier, by transforming the expression into a `case` expression. The *unboxing let to case* transformation is a variant of the transformation above, for special constructors, especially for unboxing a boxed literal. The *floating case out of let*[7] transformation floats out a `case` expression of a `let` if the value is demanded. In [Sab03b] we have shown that (ltc), (ultc) and (fcool) are $[\![\cdot]\!]$-correct if the (streval) rule of FUNDIO is a correct program transformation and strictness is defined as in Definition 2.19.

[7] [San95] calls the transformation "`case` floating from `let` right hand side"

3.4.6 Eta-expansion and -reduction

In figure 15 some rules for *eta-expansion* and *eta-reduction* are shown. In the transfor-

Eta-expansion

General rule:

$$v = \lambda x_1 \dots x_n \texttt{ -> } f\ x_1 \dots x_n \quad \overset{(\eta\text{-exp})}{\texttt{===>}} \quad v = \lambda x_1 \dots x_n \dots x_m \texttt{ -> } f\ x_1 \dots x_n \dots x_m$$

if f has arity m and $n < m$.

Restricted rule:

$$f \overset{(\text{eeta-exp})}{\texttt{===>}} \lambda x_1 \dots x_n \texttt{ -> } (f\ x_1 \dots x_n), \qquad \text{if } ar_\eta(f) = n$$

Eta case expansion:

$$\begin{array}{l} \texttt{case}\ e\ \texttt{of} \\ \quad p_1 \texttt{ -> } e_1 \\ \quad \dots \\ \quad p_n \texttt{ -> } e_n \end{array} \quad \overset{(\eta\text{-exp-case})}{\texttt{===>}} \quad \begin{array}{l} \lambda y \texttt{ -> case}\ e\ \texttt{of} \\ \quad p_1 \texttt{ -> } e_1\ y \\ \quad \dots \\ \quad p_n \texttt{ -> } e_n\ y \end{array}$$

if the following conditions hold

- e is a variable, and
- all right hand sides of the alternatives are functions, and
- all right hand sides of the alternatives are WHNFs.

Eta-reduction

$$\begin{array}{l} \texttt{let(rec)} \\ \quad f = g_1; g_1 = g_2; \dots g_k = u; \dots \\ \texttt{in}\ \lambda x_1 \dots x_n \texttt{ -> } (f\ x_1 \dots\ x_n) \end{array} \quad \overset{(\eta\text{-red})}{\texttt{===>}} \quad \begin{array}{l} \texttt{let(rec)} \\ \quad f = g_1; g_1 = g_2; \dots g_k = u; \dots \\ \texttt{in}\ f \end{array}$$

if one of the following conditions holds

(1) $u = \lambda y_1 \dots y_m \texttt{ -> } e$

(2) $u = primop$ and $ar(primop) = m$

(3) $u = c_i\ a_1 \dots a_{m'}$ and $ar(c_i) = (m + m')$

and $m \geq n$

Figure 15: Eta-expansion and -reduction

mation (η-exp) the underlying concept of arity differs from the usual. [PS94] and [San95] give an imprecise definition, by saying the used arity is the "maximum number of lambdas" of the expression, where the number of arguments is meant which can be passed to the expression, without doing "work", like evaluating a `case` or `letrec` expression. The following counter-example shows, that (η-exp) is not $[\![\cdot]\!]$-correct.

Example 3.10. *Let s and t be the following terms with $s \overset{(\eta\text{-}exp)}{\texttt{===>}} t$,*

$$s = \texttt{let}\ fun = (\lambda x_1\ x_2\ \texttt{->}\ x_1)\ (\texttt{unsafePerformIO getChar})$$
$$\texttt{in case}\ (fun\ \texttt{False})\ \texttt{of}\ \{\texttt{'a' -> 'a'; 'b' ->}\ (fun\ \texttt{False})\}$$

$$t = \texttt{let}\ fun = \lambda y\ \texttt{->}\ ((\lambda x_1\ x_2\ \texttt{->}\ x_1)\ (\texttt{unsafePerformIO getChar})\ y)$$
$$\texttt{in case}\ (fun\ \texttt{False})\ \texttt{of}\ \{\texttt{'a' -> 'a'; 'b' ->}\ (fun\ \texttt{False})\}$$

$[\![s]\!] \not\sim_c [\![t]\!]$: Let $P = \{(\mathcal{B}, \texttt{'b'})\}$, then $[\![s]\!]\Downarrow(P)$, but $\neg([\![t]\!]\Downarrow(P))$, since t requires two IO-pairs to terminate.

For understanding the (eeta-exp) transformation we define the mapping ar_η, which is similar to the function `exprArity` used in the GHC.

Definition 3.11. *$ar_\eta : L_{GHCCore} \to \mathbb{N}_0$ is defined as follows:*

$$ar_\eta(x) = \begin{cases} m, & \text{if } x \text{ is a primitive operator with arity } m \\ m, & \text{if } x \text{ a constructor with arity } m \\ 1 + ar_\eta(s) & \text{if } x = \lambda y.s \\ max\{0, ar_\eta(a) - 1\}, & \text{if } x = (a\ b) \text{ and } b \in CHEAP \\ 0, & \text{otherwise} \end{cases}$$

In [Sab03b] we have shown, that (eeta-exp) is $[\![\cdot]\!]$-correct. A variant of η-expansion for `case` expressions is (η-exp-case), but (η-exp-case) is not $[\![\cdot]\!]$-correct:

Example 3.12. *Let s, $t \in L_{GHCCore}$, where c is a constant:*

$$s := \texttt{letrec}\ z = (\texttt{unsafePerformIO getChar}); f = \lambda x\ \texttt{-> case}\ z\ \texttt{of}\ \{u\ \texttt{->}\ (\lambda w\ \texttt{->}\ w)\}$$
$$\texttt{in case}\ (f\ \texttt{True})\ \texttt{of}\ \{v\ \texttt{-> 'a'}\}$$

$$t := \texttt{letrec}\ z = (\texttt{unsafePerformIO getChar}); f = \lambda x\ \texttt{->}\ (\lambda y\ \texttt{-> case}\ z\ \texttt{of}\ \{u\ \texttt{->}\ (\lambda w\ \texttt{->}\ w)\ y\}$$
$$\texttt{in case}\ (f\ \texttt{True})\ \texttt{of}\ \{v\ \texttt{-> 'a'}\}$$

s can be transformed into t by applying the (η-case) transformation. Let $P = \emptyset$, then $[\![t]\!]\Downarrow(P)$ and $\neg([\![s]\!]\Downarrow(P))$, hence $[\![s]\!] \not\sim [\![t]\!]$.

The *eta-reduction* is defined as used in the GHC. In [Sab03b] we have shown that (η-red) is a $[\![\cdot]\!]$-correct program transformation.

3.4.7 Results

In the following theorem we remind the reader, which of the local transformations are $[\![\cdot]\!]$-correct:

Theorem 3.13. *The transformations (β-atom), (β), (flool-let), (flool-letrec), (flooacs), (dcr-let), (dcr-letrec), (uinl), (bruinl), (cheapinl), (cokc), (cokc-default), (dbe), (dae), (coe), (fcooc), (cm), (am), (fai-let), (fai-case), (cr-case), (cr-let), (η-red), (eeta-exp) are $[\![\cdot]\!]$-correct.*

The transformations (ci),(ltc),(ultc), (fcool) are $[\![\cdot]\!]$-correct if (streval) is a correct program transformation.

Proof. See [Sab03b, Theorem 4.42]. □

Therefore, these transformation can be used in the GHC for compiling programs which use `unsafePerformIO` in arbitrary contexts.

Some transformation have been shown to be not $[\![\cdot]\!]$-correct:

Theorem 3.14. *The transformations (inl), (ce), (η-exp), (η-exp-case) are not $[\![\cdot]\!]$-correct.*

Proof. See [Sab03b, Theorem 4.43]. □

These transformations have to be turned off or modified in the GHC.

[San95, Section 3.7.1] defines the transformation *constant folding* which allows to evaluate runtime-independent expressions. Constant folding seems to be $[\![\cdot]\!]$-correct, because the translated expressions can be transformed to the same expression, only by using deterministic rules of the FUNDIO calculus, which have been proven to be correct program transformations.

3.5 Global transformations

Now we give a brief overview of the global transformations, which are performed in the GHC. We yet have not investigated them in detail. In the following, at first we present a transformation, which is obviously $[\![\cdot]\!]$-correct. After that, we present three transformations, that are not $[\![\cdot]\!]$-correct. Finally we give an overview of the rest of the global transformations.

3.5.1 Correct transformations

Let floating in

This transformation[8] moves `let` bindings into expressions, but no binding outside an abstraction is moved into the body of the abstraction.

Because of the $[\![\cdot]\!]$-correctness of the (flool-let)-, (flool-letrec)-, (flooacs)- and (fai-let)-transformations, `let(rec)` bindings can be floated into other `let(rec)` bindings, into the scrutinee of a `case` expression and into an application. So it is only remaining to prove, that `let(rec)` bindings can be floated into the `case` alternatives. But this proof is easy, because we can use the (brcp) rule of the FUNDIO calculus.

[8] See [PPS96, Section 3.1], [San95, Section 5.1] and [PS98, Section 7.1].

3.5.2 Incorrect transformations

Full laziness

In contrast to let floating in, the *full laziness*[9]-transformation moves bindings out of expressions. Because the bindings are also floated out of the body of an abstraction, the transformation is not $[\![\cdot]\!]$-correct as the following counter-example shows:

Example 3.15. *Let s and t be the following expressions, where t differs only from s insofar as the binding $z =$* `unsafePerformIO getChar` *has been floated out of the abstraction.*

$$\begin{aligned} s &= \texttt{let}\ f = \lambda x\ \texttt{->}\ \texttt{let}\ z = \texttt{unsafePerformIO getChar in}\ z \\ &\quad\ \texttt{in case}\ f\ \texttt{'a' of}\ y\ \texttt{->}\ f\ \texttt{'b'} \\ t &= \texttt{let}\ f = \texttt{let}\ z = \texttt{unsafePerformIO getChar in}\ \lambda x\ \texttt{->}\ z \\ &\quad\ \texttt{in case}\ f\ \texttt{'a' of}\ y\ \texttt{->}\ f\ \texttt{'b'} \end{aligned}$$

While evaluating $[\![s]\!]$, the right hand side of f is copied for every call to f, because the right hand side is an abstraction. So, $[\![s]\!]$ needs two IO-pairs to terminate. In contrast, during the evaluation of $[\![t]\!]$ a (llet) reduction adjusts the environment insofar as the binding for z is shared for every call to f. So, $[\![t]\!]$ needs only one IO-pair to terminate. Hence, let $P = \{(\mathcal{B}, \texttt{'c'})\}$ then $[\![t]\!]\Downarrow(P)$ and $\neg([\![s]\!]\Downarrow(P))$, i.e. $[\![s]\!] \not\leq_c [\![t]\!]$.

Common subexpression elimination

Common subexpression elimination (CSE)[10] replaces identical subexpressions by a variable, and the subexpression is shared with a `let` binding.

The effect of the transformation can be reversed by using inlining and the (dcr) transformation. Because inlining is not $[\![\cdot]\!]$-correct, the same holds for CSE, which is also shown by the following counter-example:

Example 3.16. *Let s and t be the following terms, where t can be derived from s by performing CSE:*

$$\begin{aligned} s &= \texttt{case unsafePerformIO getChar of} \\ &\qquad y\ \texttt{->}\ \ \texttt{case unsafePerformIO getChar of} \\ &\qquad\qquad y'\ \texttt{-> 'a'} \\ t &= \texttt{let}\ x = \texttt{unsafePerformIO getChar in} \\ &\qquad \texttt{case}\ x\ \texttt{of} \\ &\qquad\quad y\ \texttt{->}\ \ \texttt{case}\ x\ \texttt{of} \\ &\qquad\qquad\quad y'\ \texttt{-> 'a'} \end{aligned}$$

Let $P = \{(\mathcal{B}, \texttt{'a'})\}$, then $[\![t]\!]\Downarrow(P)$, but $\neg([\![s]\!]\Downarrow(P))$.

---

[9] See. [PPS96, section 3.2], [San95, section 5.2] and [PS98, section 7.2].
[10] See [Chi98]

Static argument transformation

This transformation [San95, Section 7.1] is no longer performed in the GHC. Similar to the investigations in [PPRS00] and [PS00] for a parallel functional programming language, it is easy to show, that the *static argument transformation* is not $[\![\cdot]\!]$-correct:

Example 3.17. *Let s and t be the following terms:*

$$
\begin{array}{rcl}
s & = & \texttt{let}\ f = \lambda a\ b\ \texttt{->}\ \texttt{case unsafePerformIO getChar of} \\
 & & \qquad\qquad\qquad\quad \texttt{'d' -> 0} \\
 & & \qquad\qquad\qquad\quad y\ \texttt{->}\ f\ a\ b \\
 & & \texttt{in}\ f\ 0\ 1 \\
t & = & \texttt{let}\ f = \lambda a\ b\ \texttt{->}\ \texttt{let}\ f' = \texttt{case unsafePerformIO getChar of} \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \texttt{'d' -> 0} \\
 & & \qquad\qquad\qquad\qquad\qquad\quad y\ \texttt{->}\ f' \\
 & & \qquad\qquad\qquad\quad \texttt{in}\ f' \\
 & & \texttt{in}\ f\ 0\ 1
\end{array}
$$

s can be transformed into t by the static argument transformation, because the arguments a and b are static, i.e. they are not changed in the definition of f and they are used at the same position in the recursive call. However, the IO-multiset $P = \{(\mathcal{B}, \texttt{'d'}), (\mathcal{B}, \texttt{'e'})\}$ distinguishes $[\![s]\!]$ and $[\![t]\!]$.

3.5.3 Not yet investigated transformations

Demand analysis

The *demand analysis* is performed to obtain – beside others – strictness information (see [PP93]). Furthermore, the *constructed product result* analysis (see [BGP]) is implemented as a part of the demand analysis. Based on the obtained information the *worker/wrapper* transformation (see [PS98]) can be performed, which is implemented as a separate compiler pass.

UsageSP analysis

Based on [WP99] a type system is used, to additionally obtain information about, how often and in which context free variables occur. The advantage is that copying into a body of an abstraction is possible if it is known that this abstraction is evaluated only once, or the opposite that no copying takes place, because the abstraction is never evaluated. We yet have not investigated an according variant of the (ucp) rule, so we cannot give a statement about the $[\![\cdot]\!]$-correctness of this transformation.

Deforestation

This transformation is based on [Wad90] and used to eliminate intermediate list-like structures. An example is the expression `sum (map double) [1..n]` which is transformed to an expression, that does not use lists. More details about the implementation in the GHC can be found in [Gil96].

Specialising

The transformation described in [Jon94] generates for overloaded operators like `(+)`, special functions for every type, to avoid introducing so-called "dictionary" parameters (see [WB89]) while resolving the overloading. Another separate compiler pass is *specialising over constructors*. In [PS00] *specialising* is mentioned as problematic. These results cannot be applied easily to our semantics, as illustrated in [Sab03b].

3.5.4 Results

The most important result about our investigation of the global transformations is, that the full-laziness-transformation and the common subexpression elimination are not $[\![\cdot]\!]$-correct. They should not be performed in a FUNDIO-compatible compiler. Let-floating-in can be performed as in the GHC, because it is $[\![\cdot]\!]$-correct. The remaining global transformations are not yet investigated and should not be performed as long as they have not been proven to be correct.

4 Conclusions

We showed how to apply the calculus FUNDIO to Haskell. After representing the calculus we defined a contextual equivalence which is used to define the notion of a correct program transformation. By introducing some new transformations we enlarged the set of correct program transformations of [SS03]. This set enabled us to investigate a lot of program transformations which are performed in the Glasgow Haskell Compiler. We defined the $[\![\cdot]\!]$-correctness of program transformations on the GHC core language by introducing a translation, which translates expressions from the GHC core language to FUNDIO, and then using the correct program transformation for FUNDIO. The result is that most of the local transformations are correct in the FUNDIO sense. By turning off the few transformations that are not correct and not yet investigated transformations we achieved the prototype HasFuse – a FUNDIO-compatible modification of GHC. HasFuse allows to use `unsafePerformIO` in arbitrary contexts within Haskell programs. The behavior of these programs is no longer unpredictable, because the FUNDIO semantics gives us some predictions when and how many IO-actions will take place. From that point of view the use of `unsafePerformIO` with HasFuse is 'safe'.

5 Further work

To produce more efficient code further program transformations have to be investigated. A proof of the correctness of the (streval) transformation is necessary to complete the proofs of the $[\![\cdot]\!]$-correctness of the strictness-based transformations (ltc), (ultc) and (flcool). To perform these transformations also an investigation of the strictness analysis is necessary, we assume that a safe variant of this analysis can be developed by using an analysis based on abstract reduction as in [SSPS95, Sch00].

Another aim is to develop (and implement) correct variants of those program transformations, which have been shown to be FUNDIO-incompatible. For example the results of [Kut00] about "deterministic subexpressions" could be used to develop safe variants of inlining and common subexpression elimination.

On the other hand the now possible use of `unsafePerformIO` in arbitrary contexts should be investigated. It is possible that a declarative programming style for the IO part of a program can be integrated into Haskell.

6 Acknowledgements

I would like to thank the members of the "Glasgow Haskell Users Mailing List" and the developers of the Glasgow Haskell compiler for the useful answers to my questions about the GHC.

My special gratitude goes to Matthias Mann and Prof. Dr. Manfred Schmidt-Schauß for their constructive comments and their helpful suggestions.

References

[AFM+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246. ACM Press, 1995.

[Apt] Andrew Tolmach Apt. An external representation for the ghc core language (draft for ghc5.02). http://haskell.cs.yale.edu/ghc/docs/papers/.

[BGP] Clem Baker-Finch, Kevin Glynn, and Simon Peyton Jones. Constructed Product Result Analysis for Haskell. To appear in Journal of Functional Programming.

[Chi98] Olaf Chitil. Common subexpressions are uncommon in lazy functional languages. In Chris Clack, Kevin Hammond, and Antony J. T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 10-12, 1997, Selected Papers*, volume 1467 of *Lecture Notes in Computer Science*, pages 53–71. Springer, 1998.

[Gil96] Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Glasgow University, Department of Computing Science, 1996.

[Jon94] Mark P. Jones. Dictionary-free Overloading by Partial Evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida,* *June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 107–117, 1994.

[Kut00] Arne Kutzner. *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic choice: Operationale Semantik, Programmtransformationen und Anwendungen*. PhD thesis, J.W.Goethe-Universität Frankfurt, 2000.

[PM02] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*, 12(4&5):393–434, 2002.

[PP93] Simon L. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In J. T. O'Donnell, editor, *Glasgow Workshop on Functional Programming 1993*. Springer-Verlag, 5–7 July 1993.

[PPRS00] C. Pareja, R. Peña, F. Rubio, and C. Segura. Optimizing Eden by Transformation. In Stephen Gilmore, editor, *Trends in Functional Programming (Volume 2) . Proceedings of 2nd Scottish Functional Programming Workshop, SFP'00*, volume 2 of *Trends in Functional Programming*, pages 13–26. Intellect, 2000.

[PPS96] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 1–12. ACM Press, 1996.

[PS94] S. Peyton Jones and A. Santos. Compilation by transformation in the glasgow haskell compiler. In K. Hammond, D. N. Turner, and P. M. Sansom, editors, *Glasgow Workshop on Functional Programming*, pages 184–204, Berlin, Heidelberg, 1994. Springer.

[PS98] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

[PS00] R. Peña and C. Segura. Two non-determinism analyses in eden. Technical Report 108-00, 2000.

[Sab03a] David Sabel. *A Guide Through HasFuse*. Institut für Informatik, J. W. Goethe-Univeristät, Frankfurt, 2003. http://www.ki.informatik.uni-frankfurt.de/∼sabel/hasfuse.

[Sab03b] David Sabel. Realisierung der Ein-/Ausgabe in einem Compiler für Haskell bei Verwendung einer nichtdeterministischen Semantik. Diplomarbeit, Institut für Informatik, J.W.Goethe-Universität Frankfurt, 2003.

[San95] André Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Glasgow University, Department of Computing Science, 1995.

[Sch00] Marko Schütz. *Analysing demand in nonstrict functional programming languages*. Dissertation, J.W.Goethe-Universität Frankfurt, 2000.

[SS03] Manfred Schmidt-Schauß. FUNDIO: A lambda-calculus with a `letrec`, `case`, constructors, and an IO-interface: Approaching a theory of `unsafePerformIO`. Frank report 16, Institut für Informatik, J.W. Goethe-Universität Frankfurt am Main, 2003.

[SSPS95] Manfred Schmidt-Schauß, Sven Eric Panitz, and Marko Schütz. Strictness analysis by abstract reduction using a tableau calculus. In *Proc. of the Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 348–365. Springer-Verlag, 1995.

[The03] The GHC Team. The Glasgow Haskell Compiler User's Guide, Version 5.04, 2003. http://haskell.cs.yale.edu/ghc/docs/5.04.3/.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM Press, 1989.

[WP99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 15–28. ACM Press, 1999.