The correctness of an Optimized Code Generation

Torben Poort Lange
Computer Science Department
Aarhus University

November 1992

Abstract

For a functional programming language with a lazy standard semantics, we define a strictness analysis by means of abstract interpretation. Using the information from the strictness analysis we are able to define a code generation which avoids delaying the evaluation of the argument to an application, provided that the corresponding function is strict.

To show the correctness of the code generation, we will adopt the framework of logical relations and define a layer of predicates which finally will ensure that the code generation is correct with respect to the standard semantics.

1 Introduction

When generating code for a lazy functional programming language one often is interested in getting the most efficient code, that is avoiding expensive instructions. Consider as an example the application of a function f to an argument x. If we do not know whether the argument will be used by f, we must enclose the code for x with a *delay closure* (also called *thunk*), thus delaying the evaluation of x until needed by f. However, if an analysis shows

that the argument sooner or later will be evaluated, we might as well evaluate it before calling the function f, so that we can avoid the generation of a delay closure at all.

The information needed can be obtained from a strictness analysis [1, 6, 11], which we will define for the language. The code generation to be defined will then use this information to improve the code.

To prove the correctness of the code generation, we define a standard semantics to capture the meaning of expressions. We will then define a layer of admissible predicates that ensure that the strictness analysis is correct and that the code generated behaves as expected.

Section 2 will introduce our programming language, Section 3 defines the interpretations, Section 4 describes the framework for correctness, Section 5 proves the correctness of the strictness analysis and Section 6 proves the correctness of the code generation. Finally, Section 7 concludes with some references to related work.

This paper is an extended abstract of my M.Sc.-thesis [13], which is based on the work [16] and [18] by Hanne Riis Nielson and Flemming Nielson.

Familiarity with basic domain theoretic aspects (lattices, partial orders, chains) is assumed, as well as elementary concepts from the lambda calculus and combinatory logic.

2 The Language

The functional language will consist of a traditional typed lambda calculus to express entities of compile-time and a typed combinatory logic to express entities of run-time. Underlining will be used to denote run-time objects.

Example The expression e of type $t_1 \to t_2$ will be a compile-time function from t_1 to t_2, whereas e' of type $t'_1 \underline{\to} t'_2$ will be a run-time function from t'_1 to t'_2. When generating code we will consider expressions of the latter type. ■

The types are defined as

$$t ::= \mathbf{A}_i \mid t \times t \mid t \to t \mid \underline{\mathbf{A}}_i \mid t \underline{\times} t \mid t \underline{\to} t$$

where $\mathbf{A}_i$ are some ground types, e.g. **Bool** and **Int**.

The syntax of expressions is

$$\begin{aligned} e ::= {} & x_i[t] \mid f_i[t] \mid \langle e, e\rangle \mid \mathit{fst}\ e \mid \mathit{snd}\ e \mid \lambda x_i[t].e \mid e(e) \mid \\ & \mathit{fix}[t]\ e \mid \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e \mid F_i[t] \mid Id[t] \mid \Box[t](e,e) \\ & Tuple[t](e,e) \mid Fst[t] \mid Snd[t] \mid Cond[t](e,e,e) \mid \\ & Apply[t] \mid Curry[t]e \mid Fix[t]e \end{aligned}$$

where $f_i[t]$ denotes a primitive function such as subtraction (–[**Int**× **Int**→ **Int**]) or testing for zero ($iszero$[**Int** → **Bool**]), or constants such as integers (7[**Int**]).

The combinators are closed lambda expressions and the following informal definitions clarify our intentions:

$$*[\underline{\mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int}}] \equiv \lambda\langle x_1, x_2\rangle.x_1 * x_2$$

$$\Box[(t_0 \underline{\rightarrow} t_2) \rightarrow (t_1 \underline{\rightarrow} t_0) \rightarrow (t_1 \underline{\rightarrow} t_2)] \equiv \lambda f.\lambda g.\lambda x.f(g\ x)$$

$$Tuple[(t_0 \underline{\rightarrow} t_1) \rightarrow (t_0 \underline{\rightarrow} t_2) \rightarrow (t_0 \underline{\rightarrow} t_1 \underline{\times} t_2)] \equiv \lambda f.\lambda g.\lambda x.\langle f\ x, g\ x\rangle$$

$$Apply[((t_1 \underline{\rightarrow} t_2) \underline{\times} t_1) \underline{\rightarrow} t_2] \equiv \lambda\langle f, x\rangle.(f\ x)$$

$$Curry[((t_0 \underline{\times} t_1) \underline{\rightarrow} t_2) \rightarrow (t_0 \underline{\rightarrow} (t_1 \underline{\rightarrow} t_2))] \equiv \lambda f.\lambda x.\lambda y.f\langle x, y\rangle$$

The use of combinators will prove useful when defining an interpretation for expressions.

Example The `Miranda`™-like factorial function

```
fac n= 1, if n = 0
     = n * fac(n - 1), otherwise
```

with the argument `n` supplied at run-time is written

$$fac = \mathit{fix}(\lambda f.Cond(Iszero,\ 1,\ \Box(*,\ Tuple(Id,\ \Box(f,\ \Box(-,\ Tuple(Id,\ 1)))))))$$

where we dispense with types for readability. ■

3 Parameterized Semantics

As we want to interpret the language in different ways, it is convenient to *parameterize* the semantics upon an interpretation of the basic ingredients.

If t is a compile-time type then $[\![t]\!](\mathcal{I})$ will be the interpretation of t, where $\mathcal{I}$ is a function which interprets the run-time types. We have

$$\begin{array}{rcl} [\![\mathbf{A}_i]\!](\mathcal{I}) & = & \mathbf{A}_i \\ [\![t_1 \times t_2]\!](\mathcal{I}) & = & [\![t_1]\!](\mathcal{I}) \times [\![t_2]\!](\mathcal{I}) \\ [\![t_1 \to t_2]\!](\mathcal{I}) & = & [\![t_1]\!](\mathcal{I}) \to [\![t_2]\!](\mathcal{I}) \\ [\![t_1 \underline{\to} t_2]\!](\mathcal{I}) & = & \mathcal{I}(t_1 \underline{\to} t_2) \end{array}$$

where $\mathbf{A}_i$ will be the flat domain of a base type, $\times$ forms the cartesian product and $\to$ forms the continuous function space.

The interpretation of expressions is defined in much the same way. To handle variables we need an environment **env**, so some illustrative clauses are

$$\begin{array}{l} [\![x_i[t]]\!](\mathcal{I}) = \lambda\mathbf{env}.\mathbf{env}(x_i[t]) \\ [\![\lambda x_i[t].e]\!](\mathcal{I}) = \lambda\mathbf{env}.\lambda\mathbf{v}.([\![e]\!](\mathcal{I})\mathbf{env}[\mathbf{v}x_i[t]]) \\ [\![f_i[t]]\!](\mathcal{I}) = \lambda\mathbf{env}\mathcal{I}(f_i[t]) \\ [\![\Box[t](e_1, e_2)]\!](\mathcal{I}) = \lambda\mathbf{env}.\mathcal{I}(\Box[t])([\![e_1]\!](\mathcal{I})\mathbf{env})([\![e_2]\!](\mathcal{I})\mathbf{env}) \end{array}$$

It should be clear how to extend this definition to the whole language.

3.1 The Standard Semantics $\mathcal{S}$

This interpretation must capture the intuitive notion about the types and expressions in our language. We define

$$\begin{array}{rcl} \mathcal{S}(\underline{\mathbf{A}}_i) & = & \mathbf{A}_i \\ \mathcal{S}(t_1 \underline{\times} t_2) & = & (\mathcal{S}(t_1) \times \mathcal{S}(t_2)) \\ S(t_1 \underline{\to} t_2) & = & (\mathcal{S}(t_1) \to \mathcal{S}(t_2)) \end{array}$$

and use lifting to distinguish between e.g. the undefined pair and the pair of undefined elements.

To relate elements from a domain D with bottom element $\perp_D$ and domain $D_\perp$ with bottom element $\perp$ we define $up : D \to D_\perp$ and $dn : D_\perp \to D$ by

$$\forall d \in D \ : up(d) = d$$
$$\forall d \in D_\perp : up(d) = \begin{cases} \perp_D & \text{if } d = \perp \\ d, & \text{otherwise} \end{cases}$$

The interpretation of expressions is mostly rather straightforward. Some examples are

$$\mathcal{S}(*[t]) = up(\lambda x.x_1 * x_2 \text{ where } (x_1, x_2) = dn(x))$$

$$\mathcal{S}(\Box[t]) = \lambda g_1.\lambda g_2. \begin{cases} up(\lambda x.dn(g_1)(dn(g_2)(x))), & \text{if } g_1 \neq \perp \wedge g_2 \neq \perp \\ \perp, \text{ otherwise} \end{cases}$$

$$\mathcal{S}(Cond[t]) = \lambda g_1.\lambda g_2.\lambda g_3. \begin{cases} up(\lambda x. \begin{cases} dn(g_2)(x), \text{if } dn(g_1)(x) = true \\ dn(g_3)(x), \text{if } dn(g_1)(x) = false \\ \perp, \text{if } dn(g_1)(x) = \perp \end{cases}), \\ \qquad\qquad \text{if } g_1 \neq \perp \wedge g_2 \neq \perp \wedge g_3 \neq \perp \\ \perp, \text{ otherwise} \end{cases}$$

For the $\mathit{fix}[(t \to t) \to t]$ operator we must restrict ourself to the cases where t does not contain any underlined type constructor (t is *pure*), where t has the form $t_1 \times t_2$ but is not pure, and finally where t has the form $t_1 \underline{\to} t_2$ (t is a *frontier* type). The missing case is when $t = t_1 \to t_2$ but not pure. In this case there does not exist a general definition, but by making restrictions (e.g. on the well-formedness rules) we can avoid this type [15].

We define

$$\mathcal{S}(\mathit{fix}[(t \to t) \to t]) = \lambda G. \bigsqcup_{n \geq 0} G^n(\perp), \text{ if } t \text{ is pure}$$

$$\mathcal{S}(\mathit{fix}[(t \to t) \to t]) = \lambda G.(G_1, G_2(G_1)), \text{ if } t = t_1 \times t_1 \text{ and not pure}$$

where $G_1 = \mathcal{S}(\mathit{fix}[(t_1 \to t_1) \to t_1])(\lambda x_1.w_1 \text{ where } (w_1, w_2) = G(x_1, G_2(x_1)))$
$\quad G_2 = \lambda x_1.\mathcal{S}(\mathit{fix}[(t_2 \to t_2) \to t_2])(\lambda x_2.w_2 \text{ where } (w_1, w_2) = G(x_1, x_2))$

The latter definition arises from a version of Bekič's Theorem [2, 18].

To motivate the missing definition for frontier types $t = t_1 \underline{\to} t_2$, consider the expression $\lambda G. \sqcup_{n \geq 0} G^n(\perp)$ which is the natural definition to use for

$\mathcal{S}(\mathit{fix}[(t \to t) \to t])$. However, when G belongs to $[\![(t_1 \underline{\to} t_2) \to (t_1 \underline{\to} t_2)]\!](\mathcal{S})$ it is likely that $G(\bot) = \bot$ as we have made the interpretation strict in $\bot \in [\![t_1 \underline{\to} t_2]\!](\mathcal{S})$, so that $\sqcup_{n\geq 0} G^n(\bot) = G(\bot) = \bot$ which is undesirable. Instead, let us use the element $up(\bot)$ just above the bottom element $\bot$ and define

$$\mathcal{S}(\mathit{fix}[(t \to t) \to t]) = \lambda G.\, \sqcup_{n\geq 1}\, G^n(up(\bot)), \text{ if } t = t_1 \underline{\to} t_2$$

which is well-defined as $G(up(\bot)) = \bot$ implies $G(\bot) = \bot$ and $\mathcal{S}(\mathit{fix}[(t \to t) \to t]) = \bot$, and $G(up(\bot)) \sqsupset \bot$ implies that $(G(up(\bot)))_{n\geq 1}$ is a chain.

Example The interpretation of *fac* by $\mathcal{S}$ is **Example** The interpretation of *fac* by $\mathcal{S}$ is

$$[\![fac]\!](S) = \lambda \mathbf{env}. \bigsqcup_{n\geq 1} (\lambda \mathbf{f}.up(\lambda x. \left\{ \begin{array}{ll} 1, & \text{if } x = 0 \\ x * dn(\mathbf{f}(x-1)), & \text{if } x \neq 0 \end{array} \right\}))^n (\lambda x.\bot)$$

if $\mathcal{S}(Iszero[t]) = up(\lambda x.x = 0)$ and $\mathcal{S}(1[t]) = up(\lambda x.1)$. ■

3.2 The Strictness Analysis $\mathcal{A}$

The strictness analysis will be formulated as an abstract interpretation [1, 6]. All ground types will be interpreted by the domain $(\{\mathbf{0}, \mathbf{1}\}, \sqsubseteq)$ with $\mathbf{0} \sqsubseteq \mathbf{1}$, so we have

$$\begin{array}{rcl} \mathcal{A}(\underline{\mathbf{A}}_i) & = & \{\mathbf{0}, \mathbf{1}\} \\ \mathcal{A}(t_1 \underline{\times} t_2) & = & (\mathcal{A}(t_1) \times \mathcal{A}(t_2))_\bot \\ \mathcal{A}(t_1 \underline{\to} t_2) & = & (\mathcal{A}(t_1) \to \mathcal{A}(t_2))_\bot \end{array}$$

The interpretation of expressions is rather standard [1, 6, 11, 16] (with respect to our domains), and for a few illustrative combinators and operators we have

$$\mathcal{A}(*[t]) = up(\lambda a.a_1 \sqcap a_2 \text{ where } (a_1, a_2) = dn(a))$$

$$\mathcal{A}(\Box[t]) = \lambda s_1.\lambda s_2. \left\{ \begin{array}{ll} up(\lambda a.dn(s_1)(dn(s_2)(a))), & \\ \qquad\qquad \text{if } s_1 \neq \bot \wedge s_2 \neq \bot & \\ \bot, \quad \text{otherwise} & \end{array} \right.$$

$$\mathcal{A}(Cond[t]) =$$

$$\lambda s_1.\lambda s_2.\lambda s_3.\begin{cases} up(\lambda a.\begin{cases} dn(s_2)(a) \sqcup dn(s_3)(a), & \text{if } dn(s_1)(a) = \mathbf{1} \\ \bot & \text{if } dn(s_1)(a) = \mathbf{0} \end{cases}), \\ \qquad \text{if } s_1 \neq \bot \wedge s_2 \neq \bot \wedge s_3 \neq \bot \\ \bot, \quad \text{otherwise} \end{cases}$$

$$\mathcal{A}(fix[(t \to t) \to t]) = \lambda F. \bigsqcup_{n \geq 1} F^n(up(\bot)), \text{if} t = t_1 \underline{\to} t_2$$

Example If $\mathcal{A}(Isxero[t]) = up(\lambda a.a)$ and $\mathcal{A}(1[t]) = up(\lambda a.\mathbf{1})$ we get

$$[\![fac]\!](\mathcal{A}) = \lambda \mathbf{env}.up(\lambda a.a)$$

so that *fac* indeed is a strict function. ■

3.3 The Code Generation C

The code to be generated will be for a stack based machine. The stack contains *base values* such as booleans and integers, *pairs* of the form $\langle v_1, v_2 \rangle$, *thunks* $\{C, v\}$ to postpone the actual evaluation of C with v on the top of the stack, and *closures* $\{C; v\}$ to represent functions as data objects. We define *Val* to be the set of all stack values.

The instructions to manipulate the stack are

$$\begin{array}{ll} \mathit{ins} & ::= \texttt{const}\ b \mid \texttt{sub} \mid \texttt{mult} \mid \texttt{iszero} \mid \texttt{enter} \mid \texttt{switch} \mid \\ & \quad \texttt{branch}(C, C) \mid \texttt{tuple} \mid \texttt{fst} \mid \texttt{snd} \mid \texttt{curry}(C) \mid \texttt{apply} \mid \\ & \quad \texttt{delay}(C) \mid \texttt{resume} \mid \texttt{callrec}(l, C) \mid \texttt{call}\ l \mid \texttt{rec} \\ b & ::= \texttt{true} \mid \texttt{false} \mid \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \cdots \end{array}$$

The *instruction sequence* C is a member of *Code*, the set of all possible mstructlon sequences:

$$\mathit{Code} = \{i_1 : i_2 : \ldots : i_k \mid k \geq 1, i_j \text{ is an instruction for } 1 \leq j \leq k\} \cup \{\epsilon\}$$

The symbol $\epsilon \in Code$ denotes the empty instruction sequence.

The operational semantics is defined by the relation $\mapsto$ on configurations. Some example transitions are

$$
\begin{array}{ll}
(\texttt{const b} : C, v : ST) & \mapsto (C, \mathsf{b} : ST) \\
(\texttt{sub} : C, \langle v_1, v_2 \rangle : ST) & \mapsto (C, v_1 - v_2 : ST) \\
(\texttt{enter} : C, v : st) & \mapsto (C, v : v : ST) \\
(\texttt{switch} : C, v_1 : v_2 : ST) & \mapsto (C, v_2 : v_1 : ST) \\
(\mathsf{branch}(C_1, C_2) : C, \mathsf{true} : ST) & \mapsto (C_1 : C, ST) \\
(\mathsf{branch}(C_1, C_2) : C, \mathsf{false} : ST) & \mapsto (C_2 : C, ST) \\
(\texttt{tuple} : C, v_1 : v_2 : ST) & \mapsto (C, \langle v_1 : v_2 \rangle : ST) \\
(\texttt{delay}(C') : C, v : ST) & \mapsto (C, \{C', v\} : ST) \\
(\texttt{resume} : C, \{C', v\} : ST)) & \mapsto (C' : \texttt{resume} : C, v : ST) \\
(\texttt{resume} : C, v : ST) & \mapsto (C, ST), \text{ if } v \text{ is not a thunk} \\
(\texttt{callrec}(l, C') : C, ST) & \mapsto (C'[\texttt{callrec}(l, C')/l] : C, ST)
\end{array}
$$

The instruction sequence $C[C'/l]$ is C where every instruction "`call` l" with a free label "l" is substituted with the instruction sequence C'.

Example If

$C =$ `enter:call 1:resume:tuple:mult`
$C' =$ `resume:iszero`

then $C[C'/l] =$ `enter` : `resume` : `iszero` : `resume` : `tuple` : `mult` but $C[C'/2] = C$ as the label 2 does not occur in C.

Furthermore, $\texttt{callrec}(1, C)[C'/1] = \texttt{callrec}(1, C)$ as the label in the instruction "`call 1`" in C becomes bound, and $\texttt{callrec}(2, C)[C'/1] = \texttt{callrec}(2, C[C'/1])$ as the label 1 in C is still free. ■

With *execution sequences* we mean sequences of the form

$$\Delta = (C_0, ST_0) \mapsto (C_1, ST_1) \mapsto \cdots \mapsto (C_i, ST_i) \mapsto \cdots$$

which may be finite or infinite. We will write $\Delta(i)$ for (C_i, ST_i) and $\Delta(i \ldots)$ for the sub-sequence $(C_i, ST_i) \mapsto (C_{i+1}, ST_{i+1}) \mapsto \cdots$. Furthermore, let us for every $l \in \{*, \omega, \infty\} \cup I\!N$ define

$$
\begin{aligned}
&ExSeq(*) = \{\Delta \mid \Delta \text{ is finite}\}\\
&ExSeq(\omega) = \{\Delta \mid \Delta \text{ is infinite}\}\\
&ExSeq(\infty) = \{\Delta \mid \Delta \text{ is finite or infinite}\}\\
&ExSeq(m) = \{\Delta \in ExSeq(*) \mid \text{the length of } \Delta, \#\Delta, \text{ is } m\}\\
&ExSeq(l, C) = \{\Delta \in ExSeq(l) \mid \exists ST : \Delta(0) = (C, ST)\}\\
&ExSeq(l, C, v) = \{\Delta \in ExSeq(l) \mid \Delta(0) = (C, [v])\}
\end{aligned}
$$

Example The instruction sequence C= `enter` : `const 3` : `switch` : `tuple` : `sub` and the stack $ST = [7]$ initiates the execution sequence

$$
\begin{aligned}
\Delta = &(\texttt{enter} : \texttt{const 3} : \texttt{switch} : \texttt{tuple} : \texttt{sub}, [7]) \mapsto\\
&(\texttt{const 3} : \texttt{switch} : \texttt{tuple} : \texttt{sub}, [7, 7]) \mapsto\\
&(\texttt{switch} : \texttt{tuple} : \texttt{sub}, [3, 7]) \mapsto\\
&(\texttt{tuple} : \texttt{sub}, [7, 3]) \mapsto\\
&(\texttt{sub}, [(7, 3)]) \mapsto\\
&(\epsilon, [4])
\end{aligned}
$$

so that $\Delta \in ExSeq(5) \subseteq ExSeq(*)$ and $\Delta(5) = (\epsilon, [4])$.

Since we want to generate code for run-time functions, i.e. expressions of type $t_1 \underline{\rightarrow} t_2$, it is natural to expect $\mathcal{C}(t_1 \underline{\rightarrow} t_2) = Code_\perp$. (The bottom element $\perp$ is necessary to make *Code* a domain.) However, when coming to recursion we need fresh variables for the labels, so let us instead generate *relocatable code*, that is code from the domain $\mathbf{RelCode} = I\!N \rightarrow Code_\perp$ with the ordering $\sqsubseteq$ defined by

$$RC_l \sqsubseteq RC_2 \Longleftrightarrow \forall d \in I\!N : RC_1(d) = \perp \vee RC_1(d) = RC_2(d)$$

The type part of the code generation then is

$$\mathcal{C}(t_1 \underline{\rightarrow} t_2) = [\![t_1 \underline{\rightarrow} t_2]\!](\mathcal{A}) \times \mathbf{RelCode}$$

so that the results from the strictness analysis $\mathcal{A}$ are available.

In generating code we will observe two conditions:

A: The code makes no assumptions about whether the initial value on top of the stack is a thunk or not.

B: If the execution of the code terminates then the top of the stack will never be a thunk, and except for the top value, the stacks in the initial and final configurations will be the same.

When expressing correctness of the code generation with respect to the standard semantics we will show that these conditions are observed.

Consider the clause for $*[t]$:

$$\mathcal{C}(*[t]) = (\mathcal{A}(*[t]), \lambda d.\texttt{resume:enter:snd:resume:switch:}\\ \texttt{fst:resume:tuple:mult)}$$

The first `resume` instruction is due to condition **A**, and the `mult` instruction ensure that we do not leave a thunk on the stack (condition **B**), so that it is not necessary to terminate the instruction sequence with an additional resume instruction. This is similar to a code generation with no strictness analysis [18].

When looking at $\Box[(t_0 \underline{\rightarrow} t_2) \rightarrow (t_1 \underline{\rightarrow} t_0) \rightarrow (t_1 \underline{\rightarrow} t_2)]$ we can use the strictness information to generate slightly better code:

$$\mathcal{C}(\Box[t]) = \lambda(s_1, RC_1).\lambda(s_2, RC_2).(s, RC)$$

where $s = \mathcal{A}(\Box[t])(s_1)(s_2)$

$$RC = \lambda d. \begin{cases} \left\{ \begin{array}{l} RC_2(d) : RC_1(d), \text{if } dn(s_1(\bot) = \bot \\ \texttt{delay}(RC_2(d)) : RC_1(d), \text{ otherwise} \end{array} \right\}, \\ \qquad\qquad \text{if } RC_1(d) \neq \bot \wedge RC_2(d) \neq \bot \\ \bot, \text{ otherwise} \end{cases}$$

If the first argument expression to $\Box[t]$ is strict (i.e. $dn(s_1)(\bot) = \bot$), we can dispense with the `delay` instruction, unlike in the simple code generation.

Concerning the $\mathit{fix}[(t \rightarrow t) \rightarrow t]$ operator in the case of $t = t_1 \underline{\rightarrow} t_2$, we calculate the strictness information by ignoring the code, so that

$$\mathcal{C}(\mathit{fix}[(t \rightarrow t) \rightarrow t]) = \lambda F.(s, RC), \text{ if } t = t_1 \underline{\rightarrow} t_2$$
$$\text{where } s = \sqcup_{n \geq 1}(\lambda s'.(w_1 \text{ where } (w_1, w_2) = F(s', \cdot)))^n(up(\bot))$$

$$RC = \lambda d. \begin{cases} \texttt{callrec}(d, C_d), & \text{if } C_d \neq \bot \\ \bot, & \text{otherwise} \end{cases}$$
$$C_d = (w_2 \text{ where } (w_1, w_2) = F(s, \lambda d'.\texttt{call } d))(d+1)$$

The dot "$\cdot$" in $F(s', \cdot)$ is a shorthand for the relocatable instruction sequence $\lambda d.\bot$. We will later see that s is independent of the actual choice of instruction sequence, so that any instruction sequence would be feasible.

Example If we define $\mathcal{C}(Iszero[t]) = (up(\lambda a.a), \lambda d.\texttt{resume} : \texttt{iszero})$ and $\mathcal{C}(1[t]) = (up(\lambda a.\mathbf{1}), \lambda d.\texttt{const}\mathbf{1})$ we get

$$[\![fac]\!](\mathcal{C}) = \lambda\mathbf{env}.(up(\lambda a.a), \lambda d.\texttt{callrec}(d, C))$$

where

```
C = enter:resume:iszero:
    branch (const 1,
            enter:delay( enter:delay(const 1 ):
                         switch:delay(resume):
                         tuple:C_:call d):
            switch:delay(resume):
            tuple:C*)
C_ = resume:enter:snd:resume:switch:fst:resume:tuple:sub

C* = resume:enter:snd:resume:switch:fst:resume:tuple:mult
```

Our analysis detects the strictness of subtraction, multiplication and the recursive call, and we are thus able to avoid the generation of a `delay` instruction around the code for the first argument to all $\Box[t]$-combinators. ■

4 Correctness using Predicates

In order to express the correctness of the strictness analysis and the code generation, we will adopt the framework of *logical relations* [20, 21] and *Kripke-logical relations* [21, 22].

Definition 1 (From [17]) *An* indexed relation *over the interpretations* $\mathcal{I}_1, \ldots, \mathcal{I}_m$ *is a collection of relations*

$$\mathcal{R}_t : [\![t]\!](\mathcal{I}) \times \ldots \times [\![t]\!](\mathcal{I}_m) \to \{true, false\}$$

one for each type t. It is a logical relation *if and only if*

$$\mathcal{R}_{t_1 \to t_2}(f_1, \ldots, f_m) \equiv \forall(x_1, \ldots, x_m) : \mathcal{R}_{t_1}(x_1, \ldots, x_m) \Rightarrow \mathcal{R}_{t_2}(f_1(x_1), \ldots, f_m(x_m))$$

holds for all types t_1 and t_2.

A Kripke-indexed relation *over a non-empty partially ordered set Ω and the interpretations $\mathcal{I}_1, \ldots, \mathcal{I}_m$ is a collection of relations*

$$\mathcal{R}[\Sigma]_t : [\![t]\!](\mathcal{I}_1) \times \ldots \times [\![t]\!](\mathcal{I}_m) \to \{true,\ false\}$$

one for each type t, where

$$\forall \Sigma' \sqsupseteq \Sigma : \mathcal{R}[\Sigma]_t(x_1, \ldots, x_m) \Rightarrow \mathcal{R}[\Sigma']_t(x_1, \ldots, x_m)$$

holds for all types t. It is a Kripke-logical relation *if and only if it is a Kripke-indexed relation and*

$$\mathcal{R}[\Sigma]_{t_1 \to t_2}\ (f_1, \ldots, f_m) \equiv \forall \Sigma' \sqsupseteq \Sigma : \forall(x_1, \ldots, x_m) : \mathcal{R}[\Sigma']_{t_1}(x_1, \ldots, x_m) \Rightarrow \mathcal{R}[\Sigma']_{t_2}(f_1(x_1), \ldots, f_m(x_m))$$

holds for all types t_1 and t_2.

When coming to the *fix*$[t]$-operator it is necessary to ensure the admissibility of our predicates.

Definition 2 *A predicate $\mathcal{R}$ on the domain D is admissible if*

1. *$\mathcal{R}(\bot)$ holds.*
2. *If $(d_n)_n$ is a chain on D and $\mathcal{R}(d_n)$ holds, then $\mathcal{R}(\sqcup_n (d_n)_n)$ holds.*

To show that the relations hold we will use the principle of structural induction.

Definition 3 (From [17]) *An indexed relation $\mathcal{R}$ over $\mathcal{I}_1, \ldots, \mathcal{I}_m$ admit structural induction whenever it satisfies, that if*

$$\mathcal{R}_{t'}([\![\phi]\!](\mathcal{I}_1), \ldots, [\![\phi]\!]\mathcal{I}_m))$$

holds for all basic operators and combinators ϕ of type t' occurring in an expression e of type t, then

$$\mathcal{R}_t([\![e]\!](\mathcal{I}_1), \ldots, [\![e]\!]\mathcal{I}_m))$$

holds.

A Kripke-indexed relation $\mathcal{R}$ over Ω and $\mathcal{I}_1, \ldots, \mathcal{I}_m$ admit structural induction whenever it satisfies, that if

$$\mathcal{R}[\Sigma]_{t'}([\![\phi]\!](\mathcal{I}_1), \ldots, [\![\phi]\!]\mathcal{I}_m))$$

holds for all $\Sigma \in \Omega$ and for all basic operators and combinators ϕ of type t' occuning in an expression e of type t, then

$$\mathcal{R}[\Sigma]_t([\![e]\!](\mathcal{I}_1), \ldots, [\![e]\!]\mathcal{I}_m))$$

holds.

We now have

Fact 4 (From [17]) *Logical relations as well as Kripke-logical relations admit structural induction.*

This fact will be utilized in the following.

5 Correctness of the Strictness Analysis

To verify that the strictness information collected in $\mathcal{C}$ is correct with respect to the standard semantics, we define two predicates: $valA_t$ for t run-time and $compA_t$ for t compile-time. The definitions are quite straightforward:

$$valA_t : [\![t]\!](A) \times [\![t]\!](\mathcal{S}) \to \{true, false\}$$

$$valA_{\underline{\mathbf{A}}_i}(a, x) \equiv x \neq \bot \Rightarrow a = \mathbf{1}$$

$$valA_{t_1 \underline{\times} t_2}(a,x) \equiv x \neq \bot \Rightarrow (a \neq \bot) \wedge valA_{t_1}(a_1,x_1) \wedge valA_{t_2}(a_2,x_2)$$
where $(a_1,a_2) = dn(a), (x_1,x_2) = dn(x)$

$$valA_{t_1 \underline{\rightarrow} t_2}(s,g) \equiv g \neq \bot \Rightarrow (s \neq \bot) \wedge (\forall a,x : valA_{t_1}(a,x) \Rightarrow valA_{t_2}(dn(s)(a), dn(g)(x)))$$

We see that $\forall a : valA_t(a,\bot)$ and $\forall x : valA_t(\top,x)$ both hold, i.e. every abstract value describes the semantic value $\bot$ and every semantic value is described by the top element of the appropriate abstract domain.

$$compA_t : [\![t]\!](\mathcal{C}) \times [\![t]\!](\mathcal{S}) \rightarrow \{true, false\}$$

$$compA_{\mathbf{A}_i}(x,y) \equiv x = y$$
$$compA_{t_1 \times t_2}((x_1,x_2),(y_1,y_2)) \equiv compA_{t_1}(x_1,y_1) \wedge compA_{t_2}(x_2,y_2)$$
$$compA_{t_1 \rightarrow t_2}(F,G) \equiv \forall x,y : compA_{t_1}(x,y) \Rightarrow compA_{t_2}(F(x),G(y))$$
$$compA_{t_1 \underline{\rightarrow} t_2}((s,RC),g) \equiv valA_{t_1 \underline{\rightarrow} t_2}(s,g)$$

Lemma 5 *The clauses for compA define an admissible predicate.*

Proof A simple structural induction on the type t. ■

The correctness of the strictness analysis now amounts to showing that *compA* holds for all basic operators and combinators.

Proposition 6 *The predicate* $compA_t([\![e]\!](\mathcal{C}), [\![e]\!](\mathcal{S}))$ *holds for every expression* e *of type* t.

Proof As *compA* is a logical relation we only need to consider each combinator and operator (Fact 4). It is quite straightforward, see e.g. [13], so let us only consider the operator $fix[(t \rightarrow t) \rightarrow t]$ in the case $t = t_1 \underline{\rightarrow} t_2$.

Assume $compA_{t \rightarrow t}(F,G)$, define

$$s_n = (\lambda s'.(w_1 \text{ where } (w_1,w_2) = F(s',\cdot)))^n(up(\bot))$$
$$g_n = G^n(up(\bot))$$

and let us by induction on n show that

$$\forall RC : compA_t((s_n, RC), g_n) \qquad (P_n)$$

The base case $n = 0$ is immediate by admissibility of *compA*.

Using (P_n) and $compA_{t\to t}(F, G)$ we get $compA_t(F(s_n, \cdot), G(g_n))$, but

$$g_{n+1} = G^{n+1}(up(\bot)) = G(g_n)$$

and

$$\begin{aligned} s_{n+1} &= (\lambda s'.(w_1 \text{ where } (w_1, w_2) = F(s', \cdot)))(s_n) \\ &= w_1 \text{ where } (w_1, w_2) = F(s_n, \cdot) \end{aligned}$$

which completes the proof, since $compA_t((s_n, RC), g_n) \equiv valA_t(s_n, g_n)$ and by admissibility of *compA*. ■

As a corollary of the above proof, we see that the strictness information is independent of the code, so that the interpretation of *fix*[t] by $\mathcal{C}$, in fact, make sense.

Other approaches to correctness of a strictness analysis is [1, 4], where a suitable abstraction function is defined. The predicates, however, is just another way of defining such an abstraction function α_t: if whenever $valA_t(a, x)$ also $\alpha_t(x) \sqsubseteq a$, then α_t would respect the characteristic properties of an abstraction function.

6 Correctness of the Code Generation

The proof of correctness will consist of three layers, each described by a predicate. The first layer ensures that *fix*[t] is used correctly, the second layer additionally ensures that the generated code behaves well on the stack, and, finally, the last layer ensures the correctness of the strictness analysis and the generated code.

This approach is similar to [18], but we additionally need to incorporate the strictness analysis into the correctness predicate. This will, as we shall see, cause no difficulties.

6.1 The Substitution Property

This property is needed to guarantee that the code for $fix[t]$ will only be applied to functions that may be regarded as relocatable code with holes. First, define

$$compS'[\Sigma]_t : [\![t]\!](\mathcal{C}) \times [\![t]\!](\mathcal{C}) \to \{true, false\}$$

$$compS'[\Sigma]_{\mathbf{A}_i}(x, y) \equiv x = y$$
$$compS'[\Sigma]_{t_t \times t_2}((x_1, x_2), (y_1, y_2)) \equiv$$
$$\quad compS'[\Sigma]_{t_1}(x_1, y_1) \wedge compS'[\Sigma]_{t_2}(x_2, y_2)$$
$$compS'[\Sigma]_{t_t \to t_2}(F, G) \equiv$$
$$\quad \forall \Sigma' \supseteq \Sigma : \forall x, y : compS'[\Sigma']_{t_1}(x, y) \Rightarrow compS'[\Sigma']_{t_2}(F(x), G(y))$$
$$compS'[\Sigma]_{t_1 \underline{\to} t_2}((s_1, RC_1), (s_2, RC_2)) \equiv$$
$$\quad \forall d > max(dom(\sigma)) : compS''[\Sigma]_{t_1 \underline{\to} t_2}(RC_1(d), RC_2(d))$$
$$\quad \text{where } comps''[\Sigma]_{t_1 \underline{\to} t_2}(C_1, C_2) \equiv$$
$$\qquad (C_2 = \bot \Rightarrow C_1 = \bot) \wedge$$
$$\qquad (C_2 \neq \bot \Rightarrow (C_1 \neq \bot) \wedge C_1[\Sigma] = C_2 \wedge FreeLab(C_1) \subseteq dom(\Sigma))$$

The function $FreeLab : Code \to \{D \mid D \subseteq I\!N\}$ collects the free labels in an instruction sequence. The parameter Σ, denoting a substitution, is a set of pairs of labels and code. With $dom(\Sigma)$ we mean the set $\{l \mid (l, C) \in \Sigma\}$, and whenever $(l, C_1) \in \Sigma$ and $(l, C_2) \in \Sigma$ then $C_1 = C_2$.

Lemma 7 *The clauses for $compS'[\Sigma]$ define an admissible predicate.*

Proof A simple structural induction on the type t. ■

The desired property can now be stated as the **Substitution Property** [15, 18]:

Proposition 8 *Assume that $compS'[\Sigma]_{t \to t}(F_0, F)$ holds for every type $t = t_1 \underline{\to} t_2$ and that for $d > max(dom(\Sigma))$ and every $s \in [\![t]\!](\mathcal{A})$ we have defined*

$$C = (w_2 \text{ where } (w_1, w_2) = F(s, \lambda d'.\texttt{call } d))(d+1)$$
$$C' = \texttt{callrec}(d, C)$$
$$C'' = (w_2 \text{ where } (w_1, w_2) = F(s, \lambda d'.C'))(d+1)$$

Then

$$C \neq \bot \Rightarrow C[C'/d] = C'' \wedge FreeLab(C') \subseteq dom(\Sigma)$$

holds.

The proposition says that the "hole" in C (the instruction "`call l`" may safely be substituted with C' yielding C''.

Proof Assume $C \neq \bot$, define C_0 using F_0 in the same way as C is defined using F, and consider the two stages 1 and 2.

Stage 1:
Extend Σ to $\Sigma_1 = \Sigma \cup \{(d, \texttt{call } d)\}$. From

$$compS'[\Sigma_2]_t((s, \lambda d'.\texttt{call } d), (s, \lambda d'.\texttt{call } d))$$

we get $compS'[\Sigma_1]_t((\cdot, \lambda d'.C_0), (\cdot, \lambda d'.C))$ so that $FreeLab(C) \subseteq dom(\Sigma) \cup \{d\}$ and $C = C_0[\Sigma]$.

Stage 2:
Extend Σ to $\Sigma_2 = \Sigma \cup \{(d, C')\}$. From

$$compS'[\Sigma_2]_t((s, \lambda d'.call\ d), (s, \lambda d'.C'))$$

we get $compS'[\Sigma_2]_t((\cdot, \lambda d'.C_0), (\cdot, \lambda d'.C''))$ so that $C'' = (C_0[\Sigma])[C'/d] = C[C'/d]$ which completes the proof. ■

We only need to show that $compS'$ holds for every expression to be able to use Proposition 8:

Proposition 9 *The predicate* $compS'[\Sigma]_t([\![e]\!](\mathcal{C}), [\![e]\!](\mathcal{C}))$ *holds for every* Σ *and every expression* e *of type* t.

Proof As the predicate $compS'[\Sigma]$ is a Kripke-logical relation it is sufficient to show that $compS'[\emptyset]_t([\![e]\!](\mathcal{C}), [\![e]\!](\mathcal{C}))$ holds. For every operator and combinator other than $fix[t]$ this is straightforward, for the $fix[t]$ operator we mimic Stage 1 of Proposition 8. ■

As we only use $compS'$ with an empty substitution $\emptyset$ and identical arguments, let us define the logical relation $compS_t : [\![t]\!](\mathcal{C}) \to \{true, false\}$ by $compS_t(x) \equiv compS'[\emptyset]_t(x, x)$ and use this definition in the following.

6.2 The Well-behavedness Predicate

To ensure that the code only transforms the top element of the stack into another well-behaved element, we use the *valW* and *compW* predicates. The *valW* predicate ensures the well-behavedness of a stack element:

$$
\begin{aligned}
&valW_t : Val \to \{true, false\}\\
&valW_{\underline{\mathbf{A}}_i}(\mathsf{b}) \equiv true \text{ for all basic values } \mathsf{b} \text{ of type } \underline{\mathbf{A}}_i\\
&valW_{t_1 \underline{\times} t_2}(\langle v_1, v_2\rangle) \equiv valW_{t_1}(v_1) \wedge valW_{t_2}(v_2)\\
&valW_{t_1 \underline{\to} t_2}(\langle C, v_0\rangle) \equiv \forall v_1 : valW_{t_1}(v_1) \Rightarrow valW_{t_2}(\{C, \langle v_0, v_1\rangle\})\\
&valW_t(\{C, v\}) \equiv \forall \Delta \in ExSeq(*, C, v) : postW_t(\Delta) \wedge nothunk(\Delta)
\end{aligned}
$$

The predicate *postW* tells us whether a code sequence Δ ends with a well-behaved element on the stack, and *nothunk* ensures that the last element on the stack is not a thunk.

$$
\begin{aligned}
&postW_t, nothunk : ExSeq(m) \to \{true, false\}\\
&postW_t(\Delta) \equiv \exists v : \Delta(m) = (\epsilon, [v]) \wedge valW_t(v)\\
&nothunk(\Delta) \equiv \neg\exists C, C', v, ST : \Delta(m) = (C, \{C', v\} : ST)
\end{aligned}
$$

The definition of $valW_t(\{C, v\})$ says, that if we execute C with v on top of the stack, we end up with a well-behaved element on the stack which is not a thunk.

The well-behavedness predicate $compW_t : [\![t]\!](\mathcal{C}) \to \{true, false\}$ is defined as follows:

$$
\begin{aligned}
&compW_{\mathbf{A}_i}(x) \equiv true\\
&compW_{t_1 \times t_2}((x_1, x_2)) \equiv compW_{t_1}(x_1) \wedge compW_{t_2}(x_2)\\
&compW_{t_1 \to t_2}(F) \equiv compS_{t_1 \to t_2}(F) \wedge\\
&\qquad\qquad (\forall x : compW_{t_1}(x) \Rightarrow compW_{t_2}(F(X)))\\
&compW_{t_1 \underline{\to} t_2}((s, RC)) \equiv compSW_{t_1 \underline{\to} t_2}((s, RC)) \wedge\\
&\qquad\qquad (\forall d > 0 : compSW_{t_1 \underline{\to} t_2}(RC(d)))\\
&\quad \text{where } compSW_{t_1 \underline{\to} t_2}(C) \equiv C \neq \bot \Rightarrow (\forall v \in Val : valW_{t_1}(v) \Rightarrow\\
&\qquad\qquad valW_{t_2}(\{C, v\}))
\end{aligned}
$$

Lemma 10 *The clauses for compW define an admissible predicate.*

Proof First we must define a well-founded order $\preceq$ on types and values by

$$(t_1, v_1) \preceq (t_2, v_2) \Longleftrightarrow (t_1 \text{ is a proper subtype of } t_2) \vee \\ (t_1 = t_2 \wedge v_1 \text{ is not a thunk} \wedge v_2 \text{ is a thunk})$$

Consider then each clause for $valW_t(v)$. If v is not a thunk, then each $valW_{t'}(v')$ on the right hand side has $(t', v') \preceq (t, v)$ since t' is a subtype of t. If v is a thunk, then we have $valW_{t'}(v')$ on the right hand side with $t = t'$, but v' is not a thunk by *nothunk*, so $(t', v') \preceq (t, v)$.

As the predicate $valW$ now is well-defined, the admissibility of $compW$ is an easy structural induction on the types. ■

We are now ready to show that $compW$ holds for all operators and combinators.

Proposition 11 *The predicate* $compW_t([\![e]\!](\mathcal{C}), [\![e]\!](\mathcal{S}))$ *holds for every expression e of type t.*

Proof Even though $compW$ is not a logical relation, it is an instance of a Kripke-layered predicate which admits structural induction [17]. Therefore, for each operator or combinator, consider an execution sequence

$$\Delta \in ExSeq(m, C_1 : C_2 : \ldots : C_k)$$

and decomposeit into executionsequences $\Delta_i(m_i, C_i)$ for $i \in \{1, \ldots, k\}$. Then either use the induction hypothesis on Δ_i or write Δ_i out in detail to get the desired result. A full proof of well-behavedness can be found in [18], yet for a simpler code generation. ■

6.3 The Correctness Predicate

For run-time objects we define $valC_t : Val \times [\![t]\!](\mathcal{S}) \to \{true, false\}$ as follows:

$$valC_{\underline{\mathbf{A}}_i}(\mathsf{b}, x) \equiv valW_{\underline{\mathbf{A}}_i}(\mathsf{b}) \wedge \mathcal{B}_i[\![\mathsf{b}]\!] = x$$

$$valC_{t_1 \underline{\times} t_2}(\langle v_1, v_2 \rangle, x) \equiv \exists x_1, x_2 : x = up(x_1, x_2) \wedge$$

$$valC_{t_1}(v_1, x_1) \wedge valC_{t_2}(v_2, x_2)$$

$$\begin{aligned} valC_{t_1 \underline{\to} t_2}(\{C; v_0\}, g) \equiv\ & valW_{t_1 \underline{\to} t_2}(\{C; v_0\}) \wedge (g \neq \bot) \wedge \\ & (\forall v_1, x : valC_{t_1}(v_1, x) \Rightarrow \\ & \qquad valC_{t_2}(\{C, \langle v_0, v_1 \rangle\}, dn(g)(x))) \end{aligned}$$

$$\begin{aligned} & valC_t(\{C, v\}, x) \equiv valW_t(\{C, v\}) \wedge valWC_t(\{C, v\}, x) \\ & \quad \text{where } valWC_t \equiv \forall \Delta \in ExSeq(\infty, C, v) : \\ & \qquad (\Delta \in ExSeq(\omega) \Rightarrow x = \bot) \wedge \\ & \qquad (\Delta \in ExSeq(*) \Rightarrow postC_t(\Delta, x) \wedge nothunk(\Delta)) \end{aligned}$$

The function $\mathcal{B}_i : Val \to [\![\underline{\mathbf{A}}_i]\!](\mathcal{S})$ maps a basic value to its appropriate counterpart in the standard semantics. For an example, $\mathcal{B}_{bool}(\mathsf{true}) = true$ and $\mathcal{B}_{int}(7) = 7$. The predicate $postC$ is defined much as $postW$, the only difference being an additional parameter to the semantic value and using $valC$ instead of $valW$. We omit the details.

Finally, we define $compC_t : [\![t]\!](\mathcal{C}) \times [\![t]\!](\mathcal{S}) \to \{true, false\}$ by the following clauses:

$$\begin{aligned} & compC_{\mathbf{A}_i}(x, y) \equiv x = y \\ & compC_{t_1 \times t_2}((x_1, x_2), (y_1, y_2)) \equiv compC_{t_1}(x_1, y_1) \wedge CompC_{t_2}(x_2, y_2) \\ & compC_{t_1 \to t_2}(F, G) \equiv compW_{t_1 \to t_2}(F) \wedge compA_{t_1 \to t_2}(F, G) \wedge \\ & \qquad\qquad compWC_{t_1 \to t_2}(F, G) \\ & \quad \text{where } compWC_{t_1 \to t_2}(F, G) \equiv \\ & \qquad \forall x, y : compC_{t_1}(x, y) \Rightarrow compC_{t_2}(F(x), G(y)) \\ & compC_{t_1 \underline{\to} t_2}((s, RC), g) \equiv \\ & \quad compW_{t_1 \underline{\to} t_2}((s, RC)) \wedge compA_{t_1 \underline{\to} t_2}((s, RC), g) \wedge \\ & \quad (\forall d > 0 : compWC_{t_1 \underline{\to} t_2}(RC(d), g)) \\ & \quad \text{where } compWC_{t_1 \underline{\to} t_2}(C, g) \equiv \\ & \qquad (C = \bot \Rightarrow g = \bot) \wedge \\ & \qquad (C \neq \bot \Rightarrow g \neq \bot \wedge (\forall v, x : valC_{t_1}(v, x) \Rightarrow \\ & \qquad\qquad valCt_2(C, v, dn(g)(x)))) \end{aligned}$$

Before continuing with the proof of correctness, we must be sure the predicate is well-defined.

Lemma 12 *The clauses for compC define an admissible predicate.*

Proof Similar to the proof of Lemma 10, this is a structural induction on the type t. ■

The main theorem of the paper can now be formulated and proved.

Theorem 13 *The code generated by the optimixed code generation $\mathcal{C}$ is correct with respect to the standard semantics $\mathcal{S}$, that is the predicate $compC_t([\![e]\!](\mathcal{C}), [\![e]\!](\mathcal{S}))$ holds for every expression e of type t.*

Proof As for the *compW* predicate, *compC* is an instance of a Kripkelayered predicate [17], so that the proof is by structural induction on the operators and combinators. Here we will only consider $\Box[t]$ and $fix[t]$, which are the interesting cases. The full proof of correctness can be found in [13].

$\underline{\Box[t' \to t'' \to t] \text{ for } t' = t_0 \underline{\to} t_2, t'' = t_1 \underline{\to} t_0, t = t_1 \underline{\to} t_2.}$

Assume $compC_{t'}((s_1, RC_1), g_1)$ and $compC_{t''}((s_2, RC_2), g_2)$, define

$$s = \mathcal{A}(\Box[t' \to t'' \to t])(s_1)(s_2)$$

$$RC = \lambda d. \left\{ \begin{array}{l} \left\{ \begin{array}{l} C_2 : C_1, \text{ if } dn(s_1)(\bot) = \bot \\ \texttt{delay}(C_2) : C_1, \text{ otherwise} \end{array} \right\} \text{ if } C_1 \neq \bot \wedge C_2 \neq \bot \\ \bot, \text{ otherwise} \end{array} \right.$$

$$C_1 = RC_1(d), C_2 = RC_2(d)$$

$$g = \left\{ \begin{array}{l} up(\lambda x. dn(g_1)(dn(g_2)(x))), \text{ if } g_1 \neq \bot \wedge g_2 \neq \bot \\ \bot, \text{ otherwise} \end{array} \right.$$

and show $compC_t((s, RC), g)$.

It is, however, sufficient to choose $d > 0$, define $C = RC(d)$ and show $compWC_t(C, g)$.

The non-trivial case is $C \neq \bot$, so choose v, x with $valC_{t_1}(v, x)$. We must show that $valC_t(\{C, v\}, dn(g)(x))$ holds, so let $\Delta \in ExSeq(\infty, C, v)$.

The case $dn(s_1)(\bot) \neq \bot$:
We have $\Delta(1) = (C_1, [\{C_2, v\}])$. From $compC_{t''}((s_2, RC_2), g_2)$ we get $valC_{t_0}(\{C_2, v\}, dn(g_2)(x))$ and using $compC_{t'}((s_1, RC_1), g_1)$ we easily get $valC_{t_2}(\{C_1, \{C_2, v\}\}, dn(g)(x))$.

If $\Delta(1..) \in ExSeq(\omega)$ then $dn(g)(x) = \bot$.

If $\Delta(1..) \in ExSeq(*)$ then $postC_{t_2}(\Delta(1..), dn(g)(x)) \wedge nothunk(\Delta(1..))$ which completes the first case.

The case $dn(s_1)(\bot) = \bot$:
We have $\Delta(0) = (C_2 : C_1, [v])$. Let $\Delta_1 \in ExSeq(\infty, C_2, v)$ be the initial execution sequence of Δ. From $compC_{t''}((S_2, RC_2), g_2)$ we get

$$\begin{array}{l}\Delta_1 \in ExSeq(\omega) \Rightarrow dn(g_2)(x) = \bot) \wedge \\ \Delta_1 \in ExSeq(*) \Rightarrow postC_{t_0}(\Delta_1, dn(g_2)(x)) \wedge nothunk(\Delta_1))\end{array}$$

If $\Delta_1 \in ExSeq(\omega)$ also $\Delta \in ExSeq(\omega)$ and $dn(g)(x) = dn(g_1)(\bot) = \bot$ using $compA_{t'}((s_1, RC_1), g_1)$ and $dn(s_1)(\bot) = \bot$.

If $\Delta_1 \in ExSeq(*)$ then there exists an integer m so that $\Delta \in ExSeq(m)$ and, furthermore, $valC_{t_0}(v_1, dn(g_2)(x))$ follows for some v_1. As $\Delta(m) = (C_1, [v_1])$ we use $compC_{t'}((s_1, RC_1), g_1)$ to get

$$\begin{array}{l}(\Delta(m..) \in ExSeq(\omega) \Rightarrow dn(g)(x) = \bot) \wedge \\ (\Delta(m..) \in ExSeq(*) \Rightarrow postC_{t_2}(\Delta(m..), dn(g)(x)) \wedge \\ \qquad\qquad\qquad\qquad nothunk(\Delta(m..)))\end{array}$$

which is the desired result.

fix$[(t \to t) \to t]$ for $t = t_1 \underline{\to} t_2$.

The proof for the *fix*$[t]$-operator require a new technique. The general idea is to be able to control the number of unfoldings allowed for the `callrec` instruction. We will therefore index the instruction with a counter, which is decreased every time an unfolding take place. We extend the instruction set with $\texttt{callrec}_n$ for every $n \geq 0$, and define

$$\begin{array}{rcl}(\texttt{callrec}_0(l, C') : C, ST) & \mapsto & (\texttt{callrec}_0(l, C') : C, ST) \\ (\texttt{callrec}_{n+1}(l, C') : C, ST) & \mapsto & (C'[\texttt{callrec}_n(l, C')/l] : C, ST)\end{array}$$

so that every $\Delta \in ExSeq(\infty, \texttt{callrec}_0(l, C'))$ will be infinite.

Let us now go on to the proof.

Assume $compC_{t \to t}(F, G)$, define

$$s = \sqcup_{n \geq 1}(\lambda s'.(w_1 \text{where}(w_1, w_2) = F(s', \cdot)))^n(up(\bot))$$

$$RC = \lambda d. \begin{cases} \texttt{callrec}(d, C_d), & \text{if } C_d \neq \bot \\ \bot, & \text{otherwise} \end{cases}$$

$$C_d = (w_2 \text{ where } (w_1, w_2) = F(s, \lambda d'.\texttt{call } d))(d+1)$$

$$g = \sqcup_{n \geq 1} G^n(up(\bot))$$

and show $compC_t((s, RC), g)$. It is, however, sufficient to show that $compWC_t(RC(d),g)$ holds for every $d > 0$, but let us instead show $compC_t((s,\lambda d'.RC(d)),g)$ from which the desired property easily follows.

$C_d = \bot$ **:** We must show $compC_t((s, \lambda d.\bot), g)$, which amounts to show that $g = \bot$. The proof is in three stages:

Stage 1: Show
w_2 where $(w_1, w_2) = F(up(\bot), \lambda d'.\texttt{callrec}(d, \texttt{call } d)))(d+1) = \bot$.

Letting $\Sigma = \{(d, \texttt{callrec}(d, \texttt{call } d))\}$ we have

$$compS'[\Sigma]_t((s, \lambda d'.\texttt{call } d), (up(\bot), \lambda d'.\texttt{callrec}(d, \texttt{call } d)))$$

and using $compS_{t \to t}(F)$ and $C_d = \bot$ we complete the stage.

Stage 2 : Show $compC_t((up(\bot), \lambda d'.\texttt{callrec}(d, \texttt{call } d)), up(\bot))$.

The predicate $compS_t((up(\bot), \lambda d'.\texttt{callrec}(d, \texttt{call } d)))$ holds since $FeeLab(\texttt{callrec}(d, \texttt{call } d)) = \emptyset$.

The predicate $compW_t((up(\bot), \lambda d'.\texttt{callrec}(d, \texttt{call } d)))$ holds since every $\Delta \in ExSeq(\infty, \texttt{callrec}(d, \texttt{call } d))$ has $\Delta \in ExSeq(\omega)$.

The predicate $compA_t((up(\bot), \lambda d'.\texttt{callrec}(d, \texttt{call } d)), up(\bot))$ holds since the predicate $valT_t(\bot, \bot)$ holds.

The predicate $compC_t((up(\bot), \lambda d'.\texttt{callrec}(d, \texttt{call } d)), up(\bot))$ holds since every $\Delta \in ExSeq(\infty, \texttt{callrec}(d, \texttt{call } d))$ has $\Delta \in ExSeq(\omega)$ and $\bot(x) = \bot$ for every x.

Stage 3 : Show $g = \bot$.

Combine $compC_{t\to t}(F, G)$ and stage 2 with stage 1 to get $G(up(\bot)) = \bot$, from which $g = \bot$ follows easily.

$C_d \neq \bot$ **:** Let us by numerical induction show

$$compC_t((h^n(up(\bot)), \lambda d'.\texttt{callrec}_n(d, C_d)), G^n(up(\bot))) \qquad (P_n)$$

where $h = \lambda s'\ .(w_1 \text{ where } (w_1, w_2) = F(s', \cdot))$.

The base case, $n = 0$, amounts to show $FeeLab(\texttt{callrec}_0(d, C_d)) = \emptyset$, as every $\Delta \in ExSeq(\infty, \texttt{callrec}_0(d, C_d))$ has $\Delta \in ExSeq(\omega)$. Let $\Sigma = \{(d, \texttt{call } d)\}$ so that $compS'[\Sigma]_t(\lambda d'.\texttt{call } d, \lambda d'.\texttt{call } d)$ holds. Using $compS_{t\to t}(F)$ we get $FreeLab(C_d) \subseteq \{d\}$ implying $FreeLab(\texttt{callrec}_0 (d, C_d)) = \emptyset$.

For the induction step, we use the arguments above as well as the induction hypothesis to get

$$compW_t((h^{n+1}(up(\bot)), \lambda d'.\texttt{callrec}_{n+1}(d, C_d))) \wedge$$
$$compA_t((h^{n+1}(up(\bot)), \lambda d'.\texttt{callrec}_{n+1}(d, C_d)), G^{n+1}(up(\bot)))$$

To show $compWC_t(\texttt{callrec}_{n+1}(d, C_d)), G^{n+1}(up(\bot)))$ we use the induction hypothesis (P_n) and $compC_{t\to t}(F, G)$ to get

$$compWC_t\ ((w_2 \text{ where } (w_1, w_2) = F(h^n(up(\bot)), \lambda d'.\texttt{callrec}_n(d, C_d)))(d + 1), G^{n+1}(up(\bot)))$$

We now apply the Substitution Property to obtain

$$compWC_t(C_d[\texttt{callrec}_n(d, C_d)/d], G^{n+1}(up(\bot)))$$

from which $compWC_t(\texttt{callrec}_{n+1}(d, C_d), G^{n+1}(up(\bot)))$ follows.

This completes the proof by numerical induction.

The desired result $compC_t((s, \lambda d'.\texttt{callrec}(d, C_d)), g)$ now follows from the admissibility of $compC$.

■

7 Conclusion

For a functional programming language, we have defined a code generation which uses a strictness analysis to improve the code. Using layers of admissible predicates we were able to show the correctness of the code generation with respect to a standard semantics. This work is based on [15, 18], where the correctness of a simple code generation with no optimizations is treated. The idea of using a strictness analysis to optimize the code was presented in [16] and proved correct in [13].

Related to our approach for correctness is [15, 18], where the notion of layered predicates is used to show the correctness of a simple code generation without optimizations. The concept of layered predicates and how they interact with structural induction is discussed in [17]. A similar approach is used in [23], where structural induction is used to relate the denotational semantics for a small imperative language with an interpreter for the language.

Recent work [10, 8] translates an expression into code for a stack-based machine, such that the correctness is ensured by the "compilation" itself. The transformation is, however, based on the operational semantics of the source language.

Finally, there has been some work using domain algebra [7, 14, 24], where the denotational semantics of the source language is related to the denotational semantics of the target language using homomorphisms.

The optimization we get using a strictness analysis could be better. In [12] we compare this code generation with code generations using *strictness continuations* and *evaluation degrees*, and both remove superfluous `delay` and `resume` instructions. Strictness continuations is a way of examining the surroundings of a constructor to see if it occurs in a strict context, and evaluation degrees tells us to which extent a data constructor is evaluated.

In [13] we show the correctness of a code generation using strictness continuations for a language without higher order constructs such as $Apply[t]$ and $Curry[t]$. The correctness proof is based on layered predicates as in this paper. Furthermore, there seems to be a strong relationship between strictness continuations and evaluators [13]. An *evaluator* [4, 5] says how much evaluation must be done to an expression, which can be exploited to

generate efficient code [5, 13]. The correctness of such a code generation using evaluators might then rather easily be proved using the framework of logical relations and layered predicates instead of graph reduction as hinted in [3, 5].

The task of showing correctness of a code generation is a tiresome task, but using layered predicates we are able to divide the task into parts which can be proved one by one. Moreover, this enhances the possibility of using a mechanized verification tool such as HOL [9] or Isabelle [19].

Acknowledgement

I want to thank my supervisor, D.Sc. Flemming Nielson, who enabled me to write this paper. His comments and ideas are an invaluable part of the work. Also thanks to Torben Amtoft for proof-reading the paper. This work was supported by The Danish Research Councils under the DART-Project (grant 5.21.08.03).

References

[1] Samson Abramsky and Chris Hankin. An introduction to abstract interpretation. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1, pages 9–31. Ellis Horwood, 1987.

[2] Hans Bekič. Definable Operations in General Algebras, and the Theory of Automata and Flowcharts. *Lecture Notes in Computer Science*, Programming Languages and Their Definition(177):30–55, 1984.

[3] Geoffrey L. Burn. Using Projection Analysis in Compiling Lazy Functional Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 227–241, 1990.

[4] Geoffrey L. Burn. The Evaluation Transformer Model of Reduction and Its Correctness. *Lecture Notes in Computer Science*, TAPSOFT91: Colloquium on Combining Paradigms for Software Development(494):458–482, 1991.

[5] Geoffrey L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Research Monographs in Parallel and Distributed Computing. Pitman in association with MIT Press, 1991.

[6] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[7] Peter Dybjer. Using domain algebras to prove the correctness of a compiler. *Lecture Notes in Computer Science*, 182:98–108, 1985. Proceedings STACS 1985.

[8] Pascal Fradet and Daniel Le Mètayer. Compilation of Functional Languages by Program Transformation. *ACM Transactions on Programming Lanugages and Systems*, 13(1):21–51, 1991.

[9] Mike Gordon. HOL – A Proof Generating System for Higher-Order Logic. Cambridge CL TR 103, Computer Laboratory, University of Cambridge, 1987.

[10] John Hannan and Dale Miller. From Operational Semantics to Abstract Machines: Preliminary Results. In *ACM Conference on LISP and Functional Programming*, pages 323–332, 1990.

[11] Paul Hudak and Jonathan Young. Higher-Order Strictness Analysis in Untyped Lambda Calculus. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 97–109, 1986.

[12] Torben P. Lange. Implementation af parametriserede semantikker, 1990. Report of written project, Aarhus University, Denmark. In Danish.

[13] Torben P. Lange. Correctness of Code Generations Based on a Functional Programming Language. Master's thesis, Aarhus University, Denmark, 1992.

[14] F. L. Morris. Advice on structuring compilers and proving them correct. In *ACM Conference on Principles of Programming Languages*, pages 144–152, 1973.

[15] Flemming Nielson and Hanne R. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56:59–133, 1988.

[16] Flemming Nielson and Hanne R. Nielson. Context Information for Lazy Code Generation. In *ACM Conference on LISP and Functional Programming*, pages 251–263, 1990.

[17] Flemming Nielson and Hanne R. Nielson. Layered Predicates. In *Proceedings of the 1992 REX Workshop on "Semantics: Foundations and Applications"*, 1992. To appear in Springer Lecture Notes in Computer Science.

[18] Flemming Nielson and Hanne R. Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[19] Lawrence C. Paulson. Experience with Isabelle: A Generic Theorem Prover. Cambridge TR 143, Computer Laboratory, University of Cambridge, 1988. Preliminary version.

[20] Gordon D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, School of Artificial Intelligence, University of Edinburgh, 1973.

[21] Gordon D. Plotkin. Lambda-definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[22] J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *Proceedings in Information Processing (IFIP)*, pages 513–523. North-Holland, 1983.

[23] J. E. Stoy. The Congruence of two Programming Language Definitions. *Theoretical Computer Science*, 13:151–174, 1981.

[24] J. W. Thatcher, E. G. Wagner, and J. B. Wright. More on advice on structuring compilers and proving them correct. *Theoretical Computer Science*, 15:223–249, 1981.