Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Toward a Certified Haskell Compiler

Correctness and Implementation of the Spineless Tagless G-machine

Verified in the Coq proof assistant

Maciej Piróg

A Master Thesis under the supervision of
Dariusz Biernacki

Wrocław 2010

This thesis presents a core for a certified compiler for functional languages with lazy evaluation, like Haskell. As a preliminary, the concept of certified compilation is introduced and different semantics for lazy evaluation are discussed. The focus is on translation from a normalized lambda-calculus into instructions for a virtual machine. It is done in several steps:

- First, correctness of Peyton Jones's Spineless Tagless G-machine with respect to a natural semantics is demonstrated by a method inspired by Danvy et al.'s functional correspondence between evaluators and abstract machines.
- Then, a series of more sophisticated languages along with compilation procedures are introduced. Their natural semantics are designed to make the corresponding abstract machines more compile-friendly, that is use pointers instead of copying elements of configurations, manage one global environment instead of a series of local mappings, etc.
- Finally, the virtual machine with a set of instructions instead of program expressions is introduced.

The work embraces the most important phase of compilation, where lambda-terms are transformed into a set of imperative instructions. To complete the compiler, one needs to generate a code from the instructions in a target low-level language, but without need to refer to the intricate semantics of lazy evaluation.

All the compilation procedures between successive languages are provided and their correctness, understood as an equivalence between semantic values of source and compiled programs, is proven correct in the Coq proof assistant.

Contents

1 Introduction

This thesis addresses the issue of safety-critical compilation of lazy functional languages like Haskell [19], Clean [7] or Miranda [30]. The term 'safety-critical' means that we want to be sure that the low-level output of a compiler is semantically equivalent to the functional high-level input. The confidence in the correctness is obtained by a formal mathematical proof that has been mechanically checked by a proof system like Coq [8], HOL [21] or Twelf [40]. A program with such proof is called *certified*.

The indispensable ingredients of formal reasoning about compilers are formal definitions of the input and output languages. By a language we mean a definition of syntax and a formal semantics, like a natural semantics, an abstract machine, etc.

Additionally, a question about the definition of equivalence of semantic values arises. For example, we may expect that in a semantics for Haskell a language term evaluates to a high-level value, for example an algebraic datatype constructor, while a semantics for an assembly language will probably be a relation between two states of memory, understood as a collection of cells containing single machine words. Some relation between these two has to be defined and used in a conclusion of the correctness theorem.

Lazy languages follow two basic principles. First, they do not evaluate anything that does not have to be evaluated at the moment. In a functional setting, lazy languages share this principle with the call-by-name reduction strategy for λ-calculus, which always close the outermost β-redex. This means that arguments of a function are not evaluated at the moment of application, but their value is computed inside the body of that function only if needed.

The second principle is that each subexpression is evaluated only once (this corresponds with the call-by-need reduction strategy for λ-calculus). This does not affect the final value of evaluation, but is so crucial for performance that we need to take it into account in our formal semantics.

It is also important to stress the distance between the semantics of lazy languages and evaluation models of contemporary computers. Lazy languages offer a lot of features, like co-inductive (and thus potentially infinite) datatypes, which are not trivial to compile even in an inefficient way. This is the biggest obstacle in a way to construct a compiler (even more harsh in case of certified compilers) and the main point of our work. We start with a lazy functional language and compile it to a set of instructions for a fully imperative abstract machine. The compilation procedure is certified, that is we provide a detailed proof of its correctness, which was mechanically verified by Coq.

As the cardinal point of our work we choose an operational semantics for lazy languages proposed by Peyton Jones and Salkild, the Spineless Tagless G-machine (STG) [33]. It describes a way in which (normalized) Haskell expressions may be evaluated in an imperative way using a heap, stacks and environments. In our work, we embrace spectra on both sides of STG: first, we prove its equivalence with a higher-level natural semantics, which clearly presents the meaning of individual constructs of the language, but does not provide any means of evaluation; on the other side, we propose a series of tuned languages with semantics based on the STG evaluation model, which draw us nearer the low-level implementation (see Figure 1.1).

We finish with a virtual machine which does not operate on expressions, but has its own set of instructions. Though we do not offer a full certified compiler from Haskell to an assembly or C language, we claim that our virtual machine is completely imperative, and its further implementation does not require any insight into semantics of lazy evaluation. Thus from the point of view of semantics, the work is complete.

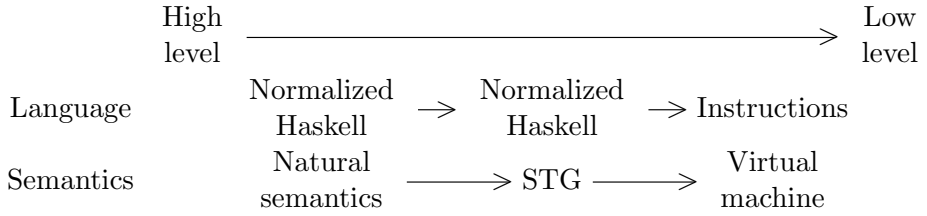


Figure 1.1: Overview of the languages and semantics

The choice of the STG machine is not accidental. It lies at the heart of the most efficient Haskell compiler, the Glasgow Haskell Compiler (GHC). In this light our work can additionally be seen as a verification of the state-of-the-art methods of compilation. Of course, the efficiency of GHC is also a result of a very neat implementation of the machine, one that cannot be repeated here, but we share some basic principles and ideas.

We would like to stress the importance of using the Coq proof assistant in this work. The level of complexity of rules for the different presented semantics and abstract machines makes the proofs on paper extremely hard to write and almost unreadable. Coq helps avoid mistakes, organize and factorize proofs, and – more importantly – frees us from verifying them by hand.

We present some languages and semantics only in Coq notation. Although definitions in Coq are more difficult to read, they are unambiguous and entirely formal. What is more, by formalization we find some optimizations.

1.1 Overview

The STG machine, and so the work presented in this thesis, is general enough to be implemented in a variety of (certified) compilers for different lazy functional languages, not only Haskell. Though, we refer to Haskell as our source language, because it is the most popular and widely studied lazy language at the moment, and most of the presented semantics and compilation methods are deeply set in research on Haskell.

To complete our study of safety-critical compilation, we perform the following four steps, which are the content of the following four chapters, respectively:

Chapter 2 (in which we find a suitable semantics) First, we need a formal definition of a language and its semantics. To do that, we refer to the previous research by Launchbury [24], Sestoft [39], Mountjoy [31], and Encina and Peña [13, 14] to realize that the good idea is to choose a language with a normalized syntax which restricts the form of subexpressions and λ-binders, so that it is easier to identify the subexpressions which should not be evaluated too eagerly (though, as shown by Ariola et al. [2], the restricted form is not necessary to incorporate a semantics similar to ours, it is an essential part of our approach to certified compilation).

We choose a language, not incidentally called STG (Shared Term Graph), proposed by Peyton Jones as a language which is evaluated by the STG machine. Though it has a very restricted form of applications and λ-binders, the simplification of Haskell expressions to STG expressions does not pose a problem. Then we propose a natural semantics for the STG language.

The next task is to formalize the language and its semantics in Coq. Our semantics is carefully designed for that purpose (for example it does not use a 'magic' generator of fresh names, which is acceptable on paper, but difficult to implement in Coq[1]). We use de Bruijn indices to obtain a nameless representation of bound variables.

Chapter 3 (in which we find a better semantics) In the second step we relate our semantics to the STG machine. They use the same expression language, so we just need to prove soundness and completeness of the STG machine with respect to our semantics.

To do that, we incorporate a method proposed by Danvy et al. [1,9], which first transforms evaluators (which can be seen as implementations of natural semantics) into the continuation-passing style [37], and then perform another transformation, defunctionalization of continuations [10, 37]. We obtain an implementation of an abstract machine, which can be identified with its formal definition. We lift this reasoning to the level of semantics and get an automatic transformation from natural semantics into abstract machines, which we call the D-CPS transformation.

[1] One would need to use more advanced techniques, like cofinite existential quantification [4].

Using a standard techniques we augment our semantics to use an additional stack for application arguments, and environments instead of substitutions. Then we perform the D-CPS transformation and get a machine which needs only a little make-up to become the Peyton Jones's STG machine. Thus, we obtain a very simple and elegant proof of correctness of the STG machine.

The STG machine is designed as a refinement of simpler machines (The G-machine [3, 23] and The Spineless G-machine [5]), but we rediscover it through the transformation, starting from a simple natural semantics.

Obviously, we formalize the whole proof in Coq. It is the first certification of such elaborate abstract machine for lazy evaluation. The D-CPS transformation turned out very Coq-friendly, because a proof of equivalence of one of our semantics and its D-CPSed version needs only a few lines of code. (Note that we do not certify the method itself, but a single case of its application).

Chapter 4 (in which we find out what our exact goal is) So far, we have been in the world of semantics. The next step is to investigate foundations of safety-critical compilation. There is a variety of approaches here, like certified and certifying compilation, refinement algebras, static analysis, proof-carrying code, or model checking (for an overview and further reference, see [16]).

We choose the method of certified and certifying compilers [17, 25, 32]. Our main reference here is a Coq-certified C compiler by Leroy [25, 26], although the interest in formal correctness of compilation procedures is as old as the theory of programming languages, to mention a compiler of arithmetic expressions by McCarthy and Painter from 1967 [28] (obviously, they did not have a working proof system, but they gave a very formal detailed reasoning on paper). There has been a vast amount of research done on the certified compilation of functional languages [6, 11, 20], but our work is the first to address lazy evaluation.

It is also important to distinguish a certified compiler from a verified compiler. The former needs an ultimately detailed proof which can be mechanically verified by a proof system, while the latter has only a proof on paper (an example is Guttman et al.'s verified implementation of the Scheme programming language [18]). In a sense, compiler verification precedes compiler certification, and so we are greatly inspired by a previous work on verification of compiling methods for lazy evaluation [12, 13, 14].

Chapter 5 (in which we implement and prove the correctness) Finally, we combine the ingredients studied in the previous steps to create a certified compiler from a normalized Haskell to the mentioned virtual machine. It can be further divided into two steps.

First, we derive another abstract machine, because we are not fully satisfied with STG. It is excellent to present a semantics as a way of computation, but it is still too

abstract (high-level) for implementations. For example, it uses environments (mappings from variables to addresses) which are modified during runtime, or stores a stack as an element of another stack. Obviously, it is possible to implement the machine directly, but the poor performance of such implementation impels us to look for better solutions.

We will not modify the STG machine explicitly, but will go back to the natural semantics, and tune the language and its semantics to use more real-life low-level abstractions. Then, we will perform the D-CPS transformation, which will bring us to a machine similar to STG to the extent of similarity of the semantics (see Figure 1.2). The machines will have the same evaluation model, that is they will perform similar transitions, but implemented in a different way.

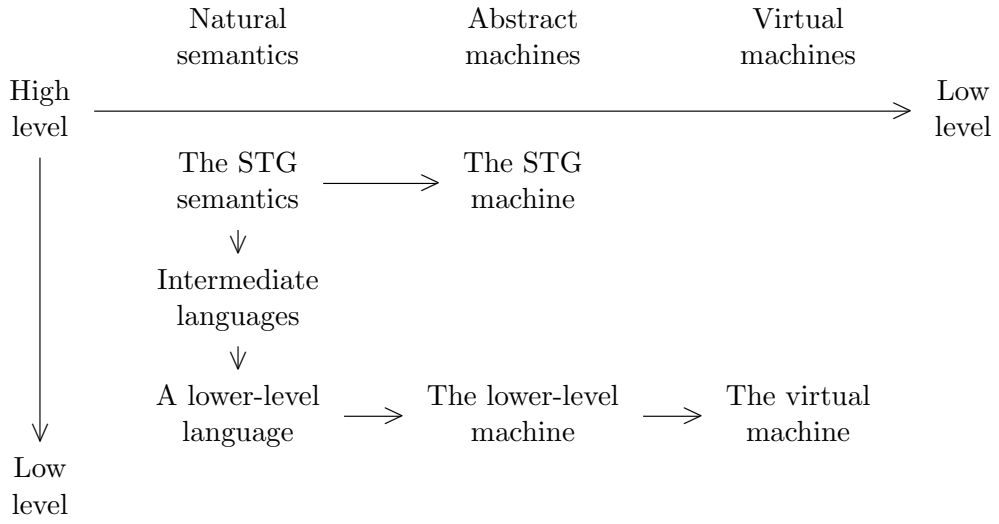


Figure 1.2: More detailed overview of the languages and semantics

Then, we propose a virtual machine (also shown in Figure 1.2). It has its own set of instructions, but, due to the specifics of lazy evaluation, it does not float far away from the language of expressions.

1.2 Related work

The amount of work done both on methods of efficient compilation of lazy languages and theoretical aspects of semantics for different lazy calculi is enormous. As a basis for our research we have chosen an article by Peyton Jones [33], which introduces the STG machine as a middle-level semantics for Haskell and a detailed description of implementation, which is based on the machine, but is not formally related to the machine

itself.

Although the paper is almost 20 years old at the moment, and the Glasgow Haskell Compiler has undergone a lot of changes in the way, and is now based on a slightly different incarnation of the machine [27], the original STG machine was a milestone in compilation techniques for lazy languages, and the further changes were too detailed to be embraced by our modest formalization in Coq.

One of our goals is to connect the implementation with the abstract machine by our virtual machine in-between. For the Coq reasoning, we could not implement all optimizations presented in the paper, either because they seemed too complicated or we decided to leave some elements on a higher level of abstraction. For example, until the very end, the heap is an abstract partial function from abstract addresses, and we could not use a pointer to the top of the heap as an offset to allocated **letrec** expressions as it is done by the original implementation of the STG machine; we have to store them on an explicit stack, as is done in the original implementation only if needed. The other side of the coin is that our virtual machine may be implemented in all memory models, including, for example, Java bytecode.

Our work is compared with the work by Encina end Peña [13, 14] which revolves around equivalence of STG-like machines (both abstract and virtual) with natural semantics. We take a different approach: we start with a different language and so a different semantics. We also use a formal method of derivation of the machines, which brings us exactly to the STG machine. Finally, we formalize our reasoning in Coq, while Encina end Peña are satisfied with proofs on paper.

This thesis shares much with an article by the author and Dariusz Biernacki [36].

1.3 Coq developement

This thesis is merely a description of what has been done in Coq. We have formalized all the presented variants of the STG language and semantics, and all theorems stating equivalence of semantics or correctness of compilation procedures are proven in Coq.

The source code is divided into four directories:

1. `stlc` contains an example of simply typed λ-calculus. It is meant to give a simple example of techniques used by the subsequent formalizations.

2. `stg` contains formalization of the STG language and a series of equivalent semantics, starting with our natural semantics, finishing with the STG machine.

3. `rpn` contains an example of certified compiler from arithmetic expressions to revere Polish notation expressions.

4. `vm` contains another formalization of the STG language and a series of semantics, finishing with the virtual machine. The compilation procedures are certified.

Note that the correctness of the STG machine is a different project then the compilation to the virtual machine, and so uses a different notations and different techniques of formalization. The former is about 7500 lines of Coq code long (ca. 100 definitions and 230 theorems) while the latter is about 3800 lines (ca. 150 definitions, 100 theorems).

2 Lazy semantics

The semantics for lazy languages is a tricky field, because we want to be able to reason about programs in a few different ways: we want to look at Haskell code as a set of equations, but on the other hand we want to include sharing of arguments; we want it to be close to mathematics, but use non-strict functions.

In this chapter we first present some intuition behind lazy evaluation, then we present a natural semantics for a normalized λ-calculus semantics. Finally, we present and formalize in Coq our semantics for a simplified Haskell – the STG language.

2.1 Haskell and lazy evaluation

As stated in the introduction, Haskell is a lazy language, which means that it does not evaluate arguments of functions and constructors until they are needed. The simplest example is:

```
loop :: a → b
loop x = loop x

firstOfThree :: a → b → c → a
firstOfThree x y z = x

result = firstOfThree 7 undefined (loop 1)
```

In the example, `undefined` breaks the evaluation and `loop` is a function that performs an infinite loop for any argument. We call the function `firstOfThree` with them as arguments, but, since the function is not interested in its second and third argument, they are not evaluated, and thus `result` is equal to `7`.

Haskell programs as recursive equations

A Haskell program may be seen as a set of recursive equations. This leads to another example, the infinite data structures. First, we define a list of `1`s:

```
list :: [Integer]
list = 1 : list
```

We may unfold the definition of `list` *ad infinitum*:

```
list = 1 : list
     = 1 : 1 : list
     = 1 : 1 : 1 : list = ...
```

Due to laziness, Haskell will not try to unfold the list all at once, and thus this definition will not stuck it in an infinite loop.

We may define a list of all natural numbers as:

```
nats = 0 : map (+1) nats
```

The definition unfolds as follows:

```
nats = 0 : map (+1) nats
     = 0 : map (+1) (0 : map (+1) nats)
     = 0 : 1 : map (+1) (map (+1) (0 : map (+1) nats))
     = 0 : 1 : map (+1) (1 : map (+1) (map (+1) nats))
     = 0 : 1 : 2 : map (+1) (map (+1) (map (+1) nats)) = ...
```

The semantics of pure functional languages understood as sets of equations was studied by Johnsson [22].

Haskell programs as DAGs

Apart from the order of evaluation, the second principle of lazy programming languages is *memoization*. It means that each value is evaluated only once and then stored, so that we will not have to compute it again if it is needed in the future.

It may be implemented using the technique named *sharing*, in which we represent expressions as directed acyclic graphs (DAGs), not as trees. In the following example assume that `h` is an expression which evaluates to the identity function (`id`):

```
twice :: (a → a) → a → a
twice f x = f (f x)

result :: Integer
result = twice h 7
```

The function `twice` uses its first arugment two times, so `result` may intuitively simplify as follows (note that `h` is evaluated twice):

```
twice h 7 → h (h 7) → ... → id (h 7) → h 7 → ... → id 7 → 7
```

But if the subexpression `h` is held as a single node in the graph, the application `twice h 7` does not create two `h`s, but only two pointers to that node, as in Figure 2.1 (where @ represents applications). Single evaluation of a node makes it updated in the whole expression:

```
twice h 7 → h (h 7) → ... → id (id 7) → id 7 → 7
```

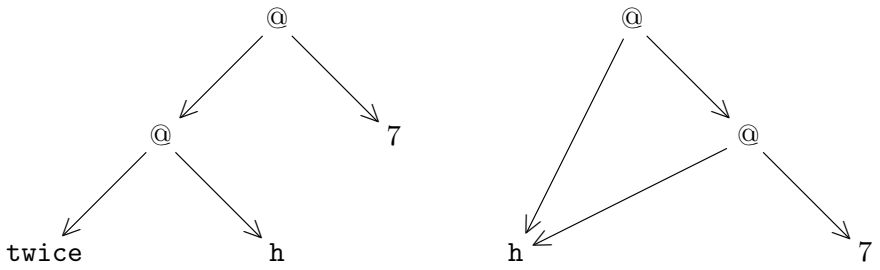


Figure 2.1: Haskell expressions as graphs

This evaluation strategy leads to a compilation technique called *graph reduction* [34], where the whole DAG of expression is represented in memory, and reduction is performed as an ordinary β-reduction. This approach was refined and implemented in a series of abstract machines: the G-machine [3, 23], the Three Instruction Machine [15], the Spineless G-machine [5], and, finally, the Spineless Tagless G-machine [33].

2.2 Natural semantics for a normalized λ-calculus

In this section we present a slightly modified[1] natural semantics by Sestoft [24], which is a refined semantics proposed by Launchbury [39] for his normalized λ-calculus. It serves us as an inspiration for construction of semantics for more complex languages.

The calculus is shown in Figure 2.2, where $\mathfrak{X} = \{x, y, p, p_0, \ldots\}$ is a set of variables. The letters e, g, w will stand for expressions.

$$e \quad \rightarrow \quad \lambda x.e \mid e\ x \mid x \mid \textbf{let}\ \overline{x_i = e_i}\ \textbf{in}\ e$$

Figure 2.2: The syntax of Launchbury's normalized λ-calculus

Notice the restricted form of applications, where arguments are only variables. As in graph reduction, they are pointers to subexpressions, which are bound by definitions in a **letrec** block.

The semantics is given in Figure 2.3. It derives judgments of the form $(\Gamma : e \downarrow \Delta : w)$. The pair $(\Gamma : e)$ is called a configuration and $(\Delta : w)$ a normal form. $\Gamma, \Delta, \Theta : \mathfrak{X} \hookrightarrow$

[1] The difference is that we do not use the explicit set of black holes. Instead, we mark the black-holed closures on the heap. The original withrawal of closures from the heap is not even possible to express in our notation.

$e \cup \{\bullet\}$ are heaps, i.e. partial functions from variables to expressions or the 'black hole' annotation. Values in the heap are called closures.

In the following, $\Gamma\{x \mapsto e\}$ stands for a heap Γ, indicating that $\Gamma(x) = e$, while $\Gamma \oplus [x \mapsto e]$ stands for a heap Γ extended or overwritten at x with e. The operation $e[\overline{x_i/p_i}]$ simultaneously substitutes each free occurrence of x_i in e with p_i.

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textsc{Lam}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.g \qquad \Delta : g[y/p] \Downarrow \Theta : w}{\Gamma : e\ p \Downarrow \Theta : w} \qquad \textsc{App}$$

$$\frac{\Gamma \oplus [p \mapsto \bullet] : e \Downarrow \Delta : w}{\Gamma\{p \mapsto e\} : p \Downarrow \Delta \oplus [p \mapsto w] : w} \qquad \textsc{Var}$$

$$\frac{\Gamma \oplus [\overline{p_i \mapsto \widehat{e_i}}] : \widehat{e} \Downarrow \Delta : w}{\Gamma : \textbf{let } \overline{x_i = e_i} \textbf{ in } e \Downarrow \Delta : w} \ (*) \qquad \textsc{Let}$$

$(*)$ $p_i \notin Dom(\Gamma)$. For all g, $\widehat{g} = g[\overline{x_i/p_i}]$

Figure 2.3: Sestoft's semantics for Launchbury's normalized λ-calculus

This semantics also implements sharing. The heap and the expression are really one graph, where nodes are stored as elements of the heap, and (free) variables are pointers. The current expression on the right-hand side of the : symbol is the element of the graph where evaluation takes place at the particular moment. When the evaluation is finished, the node is updated, which is stated in the Var rule.

When we change the node which is currently evaluated (the Var rule again) we say that we *enter* the new closure. We replace the newly entered node with a black hole. We may do that because we do not want to reenter a closure which is currently evaluated, since it would lead to an infinite loop.

The black-holing is not necessary for semantic values, that is the same judgments would be provable if we state the Var rule as follows:

$$\frac{\Gamma : e \Downarrow \Delta : w}{\Gamma\{p \mapsto e\} : p \Downarrow \Delta \oplus [p \mapsto w] : w}$$

2.3 The Shared Term Graph language (STG)

Now we present the Shared Term Graph language, which is a normalized functional language with simplified algebraic datatypes. It has a very restricted form of λ binders, which are only allowed in so called lambda-forms, which are definitions in the **letrec** expressions.

2.3.1 Syntax

The syntax of the STG language is shown in Figure 2.4, where $\mathfrak{X} = \{x, y, z, p, q, \ldots\}$ is a set of variables and $\mathfrak{C} = \{C, C_1, C_2, \ldots\}$ is a set of names of constructors. The letters e, f, g, w will stand for expressions of the STG language.

We denote sequences by juxtaposition (e.g., $x_1 \ldots x_n$) or by a line over indexed elements (e.g., $\overline{x_i}$). If not stated otherwise, sequences may be empty. Appending sequences and inserting elements is also represented by juxtaposition. The symbol ε stands for the empty sequence.

e	$\rightarrow$	$x\ \overline{x_i}$	— application
	$\mid$	$C\ \overline{x_i}$	— saturated constructor
	$\mid$	**letrec** $\overline{x_i = lf_i}$ **in** e	— local definition
	$\mid$	**case** e **of** $\overline{alt_i}$	— case expression
lf	$\rightarrow$	$\lambda_\pi\ \overline{x_i}.e$	— lambda-form
alt	$\rightarrow$	$C\ \overline{x_i} \rightarrow e$	— case alternative
π	$\rightarrow$	$u \mid n$	— update flag

Figure 2.4: The syntax of the STG language

Application We apply only single variables to tuples of variables. This limited form of application is in correspondence with the lazy evaluation: the variables are pointers to thunks representing subexpressions that will be computed only if needed (or have already been computed and updated). The tuple may be empty, so there is no need for a separate variable case in the grammar.

Constructor Constructor expressions are built using a constructor name (an element from the set $\mathfrak{C}$) and its arguments (variables). All constructors are saturated, i.e., they must be given all their arguments. Since the STG language is not typed and there are no explicit datatype definitions, we may think of constructor names as being the lowest-level identifiers, e.g., positive integers, possibly shared between datatypes. (If the STG language is used as an intermediate language in a compiler, the sharing of constructor names is not an issue, since the front-end type checking guarantees that each constructor name will be interpreted in the right datatype.)

Local definition Local definition expressions (aka **letrec** expressions) play a more significant role than in ordinary functional languages: they enclose subexpressions for lazy evaluation. Each local definition binds a lambda-form $\lambda_\pi \overline{x_i}.e$, where e is an expression and $\overline{x_i}$ is a (possibly empty) tuple of its arguments. Intuitively, we may think of it as an ordinary lambda expression. The symbol π represents an update flag (u for updatable

and n for non-updatable), which indicates whether after evaluation of the lambda-form the result should overwrite the lambda-form. If a lambda-form expects some arguments, it is already in normal form (as usual, we do not reduce under lambdas), so its update flag is always n. Definitions in one **letrec** block are assumed to be mutually recursive.

Case expression Case expressions **case** e **of** $\overline{alt_i}$ perform eager evaluation (by eager we mean 'up to the outermost constructor') of the subexpression e. The result is then matched against the list of alternatives $\overline{alt_i}$ which binds arguments of the constructor in the matched alternative. The body of the matched alternative is then evaluated according to the lazy evaluation strategy.

The transformation from an everyday-use functional language like Haskell to the STG language requires extraction of all non-variable subexpressions to **letrec** definitions, normalization of **case** expressions and a static analysis for the update-flag annotation.

2.3.2 Natural semantics

In this section, we introduce a natural operational semantics for the STG language. It uses a heap to store all the lambda-forms needed to evaluate the expression. Free variables serve as pointers to the elements of the heap. When the body of an updatable lambda-form (i.e., one with the u flag) is evaluated, it is overwritten with the value, so no expression sharing this lambda-form will evaluate the same node for a second time.

We split the set of variables $\mathfrak{X}$ into two disjoint, enumerably infinite sets: the set of bound variables $BOUND$ (ranged over by $x_1, x_2, \ldots$) and the set of heap pointers $POINTERS$ (ranged over by $p, q, p_1, q_1, \ldots$), so:

$$\mathfrak{X} = BOUND \cup POINTERS$$

It is needed to provide sound local freshness of names in the semantics, as will be discussed later on. We call an expression well-formed if and only if all its bound variables are in $BOUND$, and all of its free variables are in $POINTERS$. The semantics is designed for well-formed expressions only.

The semantics is given in Figure 2.5. It derives judgments of the form $(\Gamma : e \downarrow \Delta : f)$. The pair $(\Gamma : e)$ is called a configuration and $(\Delta : f)$ a normal form. Γ and Δ are heaps, i.e., partial functions from $\mathfrak{X}$ to LF, where LF is the set of all lambda-forms. Values in the heap are called closures.

In the following, $\Gamma\{x \mapsto lf\}$ stands for a heap Γ, explicitly indicating that $\Gamma(x) = lf$, while $\Gamma \oplus [x \mapsto lf]$ stands for a heap Γ extended or overwritten at x with lf. The operation $e[\overline{x_i/p_i}]$ simultaneously substitutes each free occurence of x_i in e with p_i.

Normal forms in this semantics are constructors and partial applications, as stated in the Con and App1 rules. (An application is partial only in the context of a heap, which encodes the whole graph of an expression.)

$$\Gamma : C\ \overline{p_i}\ \downarrow\ \Gamma : C\ \overline{p_i} \qquad \textsc{Con}$$

$$\Gamma\{p \mapsto \lambda_n\ x_1 \ldots x_m.e\} : p\ p_1 \ldots p_n\ \downarrow\ \Gamma : p\ p_1 \ldots p_n \quad \text{where } n < m \qquad \textsc{App1}$$

$$\frac{\Gamma : e[x_1/p_1 \ldots x_m/p_m]\ \downarrow\ \Delta : w}{\Gamma\{p \mapsto \lambda_n\ x_1 \ldots x_m.e\} : p\ p_1 \ldots p_m\ \downarrow\ \Delta : w} \qquad \textsc{App2}$$

$$\frac{\begin{array}{c}\Gamma : e[x_1/p_1 \ldots x_m/p_m]\ \downarrow\ \Delta : q\ q_1 \ldots q_k \\ \Delta : q\ q_1 \ldots q_k\ p_{m+1} \ldots p_n\ \downarrow\ \Theta : w\end{array}}{\Gamma\{p \mapsto \lambda_n\ x_1 \ldots x_m.e\} : p\ p_1 \ldots p_n\ \downarrow\ \Theta : w}\ m < n \qquad \textsc{App3}$$

$$\frac{\Gamma : e\ \downarrow\ \Delta : C\ \overline{q_i}}{\Gamma\{p \mapsto \lambda_u.e\} : p\ \downarrow\ \Delta \oplus [p \mapsto \lambda_n.C\ \overline{q_i}] : C\ \overline{q_i}} \qquad \textsc{App4}$$

$$\frac{\begin{array}{c}\Gamma : e\ \downarrow\ \Delta\{q \mapsto \lambda_n\ x_1 \ldots x_k\ x_{k+1} \ldots x_n.f\} : q\ q_1 \ldots q_k \\ \Delta \oplus [p \mapsto \lambda_n\ x_{k+1} \ldots x_n.f[x_1/q_1 \ldots x_k/q_k]] : q\ q_1 \ldots q_k\ p_1 \ldots p_m\ \downarrow\ \Theta : w\end{array}}{\Gamma\{p \mapsto \lambda_u.e\} : p\ p_1 \ldots p_m\ \downarrow\ \Theta : w} \qquad \textsc{App5}$$

$$\frac{\Gamma \oplus \overline{[p_i \mapsto lf_i[\overline{x_i/p_i}]]} : e[\overline{x_i/p_i}]\ \downarrow\ \Delta : w}{\Gamma : \textbf{letrec}\ \overline{x_i = lf_i}\ \textbf{in}\ e\ \downarrow\ \Delta : w}\ \overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma) \qquad \textsc{Letrec}$$

$$\frac{\Gamma : e\ \downarrow\ \Delta : C_k\ \overline{p_j} \quad \Delta : e_k[\overline{x_{kj}/p_j}]\ \downarrow\ \Theta : w}{\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}\ \downarrow\ \Theta : w} \qquad \textsc{Case}$$

Figure 2.5: The natural semantics of the STG language

There are two more rules for applications of variables representing non-updatable closures: App2, when there are just enough arguments, and App3, when there are too many arguments. Intuitively, we evaluate the body of the lambda-form, substituting actual arguments for formal arguments. If there are too many arguments, we use only the prefix of the argument list of the appropriate length. If the closure evaluates to a partial application, we append the remaining suffix of the argument list and continue with evaluation. If it evaluates to a constructor, the whole expression does not have a normal form, since it would be an application of the constructor to the suffix of the argument list, which is already saturated; such expression would be ill-typed in any strongly typed language.

The rules for applications of variables representing updatable closures are similar. For a variable p representing an updatable (thus argument-free) closure, if the closure evaluates to a constructor (App4), it is the value of the expression. But we also need to update the heap with the constructor, so that if any other expression in some lambda-

form in the heap uses the pointer p, the closure will not have to be evaluated again. If the closure evaluates to a partial application $q\ q_1 \dots q_k$ (App5), we update the closure under the pointer p with the lambda-form under the pointer q, but with first k arguments already fed with $q_1 \dots q_k$. (In the App5 rule $n > k$, since $q\ q_1 \dots q_k$ is a partial application.)

Variables, addresses, and fresh pointers

A variable is fresh if and only if it does not interfere with any other variable in the derivation tree by an undesired variable capture. The freshness check (sometimes called a generation of a fresh variable) is local iff it can be done using only the context of a single rule, and does not refer to the whole derivation tree or any kind of external 'fresh names generator'. Locality is a desirable property when one wants to reason in low-level details necessary for an implementation or formalization in proof systems like Coq.

Our solution with a bipartite set of variables solves the problem: generating fresh addresses in the Letrec rule is local (we need freshness only with respect to the heap) and it fits the design pattern of nameless bound variables representation in Coq, where bound variables are represented as de Bruijn indices, and free variables as atoms.

In order to formalize the above intuitions, we use the following definitions:

Definition 1. *Let e be an expression or a lambda-form, and Γ be a heap. Then:*

1. *e is* well-formed *iff its bound variables are in BOUND and its free variables are in POINTERS.*
2. *e is* closed by *a heap Γ iff all its free variables are in $Dom(\Gamma)$.*
3. *Γ is* well-formed *iff $Dom(\Gamma) \subseteq$ POINTERS and each closure in Γ is well-formed and closed by Γ.*
4. *The configuration $(\Gamma : e)$ is* well-formed *iff Γ and e are well formed and e is closed by Γ.*

We do not mind that ill-formed programs and configurations may evaluate to nonsense values. For example the configuration $(\emptyset : \textbf{letrec}\ x = \lambda_n.C\ \textbf{in}\ p)$ may evaluate to C if the lambda-form in the **letrec** expression is allocated under the address p.

The following theorem ensures that if the root configuration is well-formed, configurations are well-formed throughout the proof and no variables are captured:

Theorem 2. *For a well-formed configuration $(\Gamma : e)$, if $(\Gamma : e \downarrow \Delta : w)$, then all configurations and normal forms in the proof of $(\Gamma : e \downarrow \Delta : w)$ (including $\Delta : w$) are well-formed and all substitutions replace pointers for bound variables.*

Proof. By a simple induction on derivation trees. □

2.3.3 Formalization in Coq

Representation on an example

Since the STG language has a very restricted form, it is not easy to find short and simple examples to isolate individual concepts of the formalization (for example, binders in lambda-forms are always paired with a list of lambda-forms in a **letrec** expression, and alternatives always go in lists, so there is no simple expression to be an example for the formalization of nested binders). Thus, we decided to present them on an example of the simply-typed lambda calculus (STLC), the syntax being:

$$e \;\rightarrow\; x \mid \lambda x.e \mid e\ e$$

Binders We use de Bruijn indices to represent bound variables, that is natural numbers indicating how many binders there are on a path from a variable to the corresponding (binding that variable) lambda. For example the term

$$\lambda x.\lambda y.(x\ (\lambda z.z\ x\ y))\ x$$

may be presented with de Bruijn indices as:

$$\lambda.\lambda.(1\ (\lambda.0\ 2\ 1))\ 1$$

The STLC language may be encoded in Coq as follows:

```
Inductive term : Set :=
| var : nat → term
| abs : term → term
| appl : term → term → term.

Coercion var : nat >-> term.
```

Most functions and predicates that operate inductively on expressions have the `offset` argument, which is the number of lambdas 'above' the current subexpression. For example, consider the predicate `closed` which is satisfied for those expressions in which every variable has a corresponding binder:

```
Inductive closed (offset : nat) : term → Prop :=
| closed_var : forall v : nat, v < offset →
    closed offset (var v)
| closed_abs : forall t : term, closed (S offset) t →
    closed offset (abs t)
| closed_app : forall t1 t2 : term, closed offset t1 →
    closed offset t2 →
    closed offset (appl t1 t2).
```

Substitutions We substitute only not bound variables (those without corresponding binders) by closed terms. The type of the substitution is `nat → option term`. The effect of substitution should be as in the following example:

$$(0\ (\lambda.0\ 2\ 1)\ 1)\ [0 \mapsto E_1;\ 1 \mapsto E_2] = E_1\ (\lambda.0\ E_2\ E_1)\ E_2$$

For each variable, the current offset is subtracted from the index, which results in the global index of the variable, not in the context of surrounding binders (the global index is the one in the domain of the substitution). We do not worry about the offsets of free variables in terms in the substitution, since we assume that they are closed.

```
Definition apply (offset : nat) (f : nat → option term)
  (v : nat) : term :=
if le_lt_dec offset v
  then match f (v - offset) with
    | Some t ⇒ t
    | _ ⇒ v
    end
  else var v.
```

Now, it suffices to inductively propagate `apply` on the whole term:

```
Fixpoint subst (offset : nat) (f : nat → option term) (t : term)
  {struct t} : term :=
match t with
| var v ⇒ apply offset f v
| abs t ⇒ abs (subst (S offset) f t)
| appl t1 t2 ⇒ appl (subst offset f t1) (subst offset f t2)
end.
```

Environments An environment is a list which contains some information about (and, in a sense, binds) a free (i.e. without corresponding binder) variable. The position in the list corresponds to the index of the free variable. For example, consider the following pair of a typing environment and a term:

$$[x : T_1,\ y : T_2] \quad (\lambda z.z\ x\ y)\ x\ y$$

In the de Bruijn notation it would become:

$$[T_1, T_2] \quad (\lambda.0\ 1\ 2)\ 0\ 1$$

Types and the typing rules may be encoded in Coq as follows:

```
Inductive type : Set :=
| type_base : type
| type_arrow : type → type → type.

Infix "⇛" := type_arrow (at level 70, right associativity).

Definition env := list type.

Inductive typing : env → term → type → Prop :=
| typing_var : forall (e : env) (v : nat) (s : type),
    nth_error e v = Some s → typing e (var v) s
| typing_abs : forall (e : env) (t : term) (s1 s2 : type),
    typing (s1 :: e) t s2 → typing e (abs t) (s1 ⇛ s2)
| typing_app : forall (e : env) (t1 t2 : term) (s1 s2 : type),
    typing e t1 (s1 ⇛ s2) → typing e t2 s1 →
    typing e (appl t1 t2) s2.
```

In the `typing_var` constructor we look up a type in the environment, while in `typing_abs` we extend the environment.

Note that such environments are very fragile, and have to work in a perfect harmony with terms, since the number of elements in an environment is the offset of the associated term.

Later, in our formalization of the STG machine we represent each closure on the heap by a pair of a lambda-form and an environment, which binds addresses to free variables of the term, for example:

$$(\lambda xy.\ x\ y\ p\ r,\ [p \mapsto a, r \mapsto b])$$

It may be represented as:

$$(\lambda[2].\ 0\ 1\ 2\ 3,\ [a, b])$$

But we want to store *only* the free variables, so if the term is encoded as

$$\lambda[2].\ 0\ 1\ 3\ 6$$

then our notion of environments is not sufficient. Thus, we reify the indices in the environment. The basic feeling of the environments remains the same, but we may miss some values, so:

$$(\lambda[2].\ 0\ 1\ 3\ 6,\ [1 \mapsto a, 4 \mapsto b])$$

The disadvantage is that previously we only had to push some elements to the front of the environment while opening a term, for example the Abs typing rule:

$$\frac{s_1, \Gamma \vdash e : s_2}{\Gamma \vdash \lambda\, e : s_1 \to s_2}$$

Now, we have to take care of the indices:

$$\frac{0 \mapsto s_1, 1 \mapsto t_1, \ldots, (n+1) \mapsto t_n \vdash e : s_2}{0 \mapsto t_1, \ldots, n \mapsto t_n \vdash \lambda\, e : s_1 \to s_2}$$

We call the operation of transforming $0 \mapsto t_1, \ldots, n \mapsto t_n$ into $1 \mapsto t_1, \ldots, (n+1) \mapsto t_n$ *shifting*.

The STG language in Coq

First, we introduce another kind of variables, *atoms*, defined as natural numbers:

```
Definition atom := nat.

Inductive var : Set :=
| Index : nat → var
| Atom : atom → var.
```

The `Index` variables represent the set *BOUND*, while the `Atom` variables represent the set *POINTERS*.

The language is defined as follows:

```
Definition constructor := nat.
Definition bind := nat.

Inductive upd_flag : Set :=
| Update : upd_flag
| Dont_update : upd_flag.

Inductive expr : Set :=
| App : var → vars → expr
| Constr : constructor → vars → expr
| Letrec : list lambda_form → expr → expr
| Case : expr → list alt → expr
with lambda_form : Set :=
| Lf : upd_flag → bind → expr → lambda_form
with alt : Set :=
| Alt : constructor → bind → expr → alt.
```

The arguments of multiple binders counts arguments from the right to the left, for example:

$$\lambda_n\ x\ y\ z.\ x\ y\ z$$

in de Bruijn notation becomes:

$$\lambda_n\ [3].2\ 1\ 0$$

Note that the `Letrec` constructor implicitly binds all the lambda forms inside (0 for the first element). For example, the following expression:

$$\begin{array}{rl} \textbf{letrec} & f = \lambda_n.\ g\ f \\ & g = \lambda_n\ x\,y.\ f\ g\ x\ y \\ \textbf{in} & g\ f\ f \end{array}$$

may be translated to the following de Bruijn form:

$$\begin{array}{rl} \textbf{letrec} & \lambda_n[0].\,1\,0 \\ & \lambda_n[2].\,2\,3\,1\,0 \\ \textbf{in} & 1\,0\,0 \end{array}$$

It can be encoded in Coq as follows:

```
Letrec (Lf n 0 (App (Index 1) (Index 0 :: nil) ::
        Lf n 2 (App (Index 2)
                    (Index 2 :: Index 3 :: Index 1 ::
                    Index 0 :: nil)) ::
        nil)
       (App (Index 1) (Index 0 :: Index 0 :: nil)
```

The natural semantics

The heap in Coq is, of course, a partial function. We also define some basic operation with obvious semantics:

```
Definition heapA := var → option lambda_form.

Definition setA (g : heapA) (a : var) (lf : lambda_form)
  heapA := (...)
Definition allocA (g : heapA) (ats : vars) (lfs : defs) :
  heapA := (...)
```

Though the semantics does not use environments, we need it defined for use of substitution:

```
Definition env : Set := list (nat * var).

Definition subst_var (nas : env) (v : var) : var :=
match v with
| Index k ⇒
  match find_assoc nas k with
  | Some a ⇒ a
```

```
  | None ⇒ v
  end
| Atom a ⇒ v
end.

Fixpoint map_expr (op : var → var) (offset : nat) (e : expr)
  {struct e} : expr := (...)

Definition subst (xs : env) (e : expr) : expr :=
map_expr (subst_var xs) 0 e.

Notation " e ~ [ xs ] " := (subst xs e) (at level 40).
```

The `zip_var_list` function takes a list of variables and a number, and makes an environment with keys descending, starting from the predecessor of the number in the argument, for example:

```
zip_var_list 6 [Index 2, Atom 5, Index 7] =
[(5, Index 2), (4, Atom 5), (3, Index 7)]]
```

Its definition is straightforward:

```
Fixpoint zip_var_list (n : nat) (ats : list var) {struct ats}
  : (env) :=
match ats with
| a :: ts ⇒ (n - 1, a) :: zip_var_list (n - 1) ts
| nil ⇒ nil
end.
```

Now, when we have a notion of substitutions we can simply rewrite the definition of semantics in Coq:

```
Reserved Notation "($ a $ b ↓ c $ d )"
                  (at level 70, no associativity).

Inductive STG : heapA → expr → heapA → expr → Prop :=

| Con : forall Gamma C vs,
  ($Gamma $ Constr C vs ↓ Gamma $ Constr C vs)

| App1 : forall Gamma p pn m e,
  Gamma p = Some (Lf_n m e) →
  m > length pn →
```

```
  ($ Gamma $ App p pn ↓ Gamma $ App p pn)

| App2 : forall Gamma Delta p e pn m w,
  Gamma p = Some (Lf_n m e) →
  m = length pn →
  ($ Gamma $ e~[zip_var_list m pn] ↓ Delta $ w) →
  ($ Gamma $ App p pn ↓ Delta $ w)

| App3 : forall Gamma Delta Theta q qk p e pm m w,
  Gamma p = Some (Lf_n m e) →
  m < length pm →
  ($ Gamma $ e~[zip_var_list m (firstn m pm)] ↓ Delta
                                              $ App q qk) →
  ($ Delta $ App q (qk ++ skipn m pm) ↓ Theta $ w) →
  ($ Gamma $ App p pm ↓ Theta $ w)

| App4 : forall Gamma Delta e p C qs,
  Gamma p = Some (Lf_u e) →
  ($ Gamma $ e ↓ Delta $ Constr C qs) →
  ($ Gamma $ App p nil ↓ setA Delta p (Lf_n 0 (Constr C qs)) $
                                          Constr C qs)

| App5 : forall Gamma Delta Theta p pn q qk e f n w,
  Gamma p = Some (Lf_u e) →
  Delta q = Some (Lf_n n f) →
  length qk < n →
  ($ Gamma $ e ↓ Delta $ App q qk) →
  ($ setA Delta p (Lf_n (n - length qk) (f~[zip_var_list n qk]))
    $ App q (qk ++ pn) ↓ Theta $ w) →
  ($ Gamma $ App p pn ↓ Theta $ w)

| Let : forall Gamma Delta lfs e w ats,
  length ats = length lfs →
  are_atoms ats →
  (forall a : var, In a ats → Gamma a = None) →
  ($ allocA Gamma ats
      (map (subst_lf (zip_var_list (length lfs) ats)) lfs)
    $ e~[zip_var_list (length lfs) ats]
    ↓ Delta $ w) →
  ($ Gamma $ Letrec lfs e ↓ Delta $ w)
```

```
| Case_of : forall Gamma Delta Theta b e e0 als w c c0 ps,
  length ps = b →
  select_case als c = Some (Alt c0 b e0) →
  ($ Gamma $ e ↓ Delta $ Constr c ps) →
  ($ Delta $ e0∼[zip_var_list b ps] ↓ Theta $ w) →
  ($ Gamma $ Case e als ↓ Theta $ w)

where "($ a $ b ↓ c $ d )" := (STG a b c d).
```

3 Correctness of the STG machine

In this chapter we rediscover the STG machine: we perform a series of transformation on our natural semantics to get a machine which turns out to be almost the Spineless Tagless G-machine (we need only to change some notation and merge two transition rules).

3.1 Functional correspondence

In this section we describe a method that facilitates a mechanical derivation of abstract machines from natural semantics. It is inspired by functional correspondence that consists in first, transforming an evaluator in direct style that implements a natural semantics into continuation-passing style (CPS) and second, defunctionalizing the continuations of the CPS evaluator, which leads to an evaluator implementing an abstract machine [1]. We illustrate the functional correspondence with the example of evaluating arithmetic expressions.

Let $\Sigma = \{+, *, -, \ldots\}$ be a set of binary symbols and $[\cdot]\colon \Sigma \to \mathbb{Z}^{\mathbb{Z}\times\mathbb{Z}}$ be a natural interpretation of symbols in Σ. For any $\mathbf{n} \in \mathbb{Z}$ let $|\mathbf{n}|$ denote its absolute value. The syntax and semantics of arithmetic expressions are shown in Figure 3.1.

$$e \;\to\; \mathbf{n} \mid \mathbf{abs}\ e \mid e \odot e \quad \text{where } \mathbf{n} \in \mathbb{Z} \text{ and } \odot \in \Sigma$$

$$\mathbf{n} \Downarrow \mathbf{n} \qquad \frac{e \Downarrow \mathbf{n}}{\mathbf{abs}\ e \Downarrow |\mathbf{n}|} \qquad \frac{e_1 \Downarrow \mathbf{n_1} \quad e_2 \Downarrow \mathbf{n_2}}{e_1 \odot e_2 \Downarrow \mathbf{n_1}[\odot]\mathbf{n_2}}$$

Figure 3.1: Arithmetic expressions – the syntax and natural semantics

It is straightforward to implement an evaluator for this semantics in a functional metalanguage, i.e., to write a function `eval` such that `eval e` $= \mathbf{n}$ iff the judgment $(e \Downarrow \mathbf{n})$ is provable. For each semantic rule, the function is recursively called and the final result is obtained by applying the corresponding operation to the results of the recursive calls. It could be encoded in Haskell as follows:

```
data Expr = Const Integer
          | Abs Expr
          | Op Expr String Expr

interp :: String → Integer → Integer → Integer
interp "+"   = (+)
interp "*"   = (*)
interp "-"   = (-)
interp "mod" = mod

eval :: Expr → Integer
eval (Const n)       = n
eval (Abs e)         = abs n
  where n = eval e
eval (Op e1 op e2)   = interp op n1 n2
  where n1 = eval e1
        n2 = eval e2
```

In the next phase we transform the evaluator into CPS. Now, the evaluator has one more argument – a continuation. The evaluator no longer returns a value, instead it tail-calls itself or applies the continuation to a value. To compute the value of an expression, one supplies the evaluator with the identity continuation (`kId`):

```
evalCps :: Expr → (Integer → a) → a
evalCps (Const n)      k = k n
evalCps (Abs e)        k = evalCps e (λn → k (abs n))
evalCps (Op e1 op e2) k = evalCps e1
  (λn1 → evalCps e2 (λn2 → k (interp op n1 n2)))

kId :: Integer → Integer
kId = id
```

The next step is the defunctionalization of continuations. Each construction of a continuation (either by a named value, like `kId`, or anonymously, like `λn → k (abs n)`) is replaced by an explicit closure, which stores all the free variables of the continuation. Each application of a continuation is replaced by an application of the function `apply` which takes the closure as an argument and evaluates accordingly:

```
data Cont a = Id
            | K1 (Cont a)
            | K2 Expr String (Cont a)
            | K3 Integer String (Cont a)

apply :: Cont Integer → Integer → Integer
apply Id            n  = n
apply (K1 k)        n  = apply k (abs n)
apply (K2 e2 op k) n1 = evalDcps e2 (K3 n1 op k)
apply (K3 n1 op k) n2 = apply k (interp op n1 n2)

evalDcps :: Expr → Cont Integer → Integer
evalDcps (Const n)     k = apply k n
evalDcps (Abs e)       k = evalDcps e (K1 k)
evalDcps (Op e1 op e2) k = evalDcps e1 (K2 e2 op k)
```

Note that the `Cont` datatype behaves like a stack, with `Id` corresponding to the empty stack, and `K1`, `K2` and `K3` to three kinds of its elements.

The mutually recursive functions `evalDcps` and `apply` may be thought of as evaluators of two semantics defined in terms of each other: $\mathcal{E}$, proving judgments of the form $\mathcal{E}\langle e, \overline{K_i}\rangle \nwarrow \mathbf{n}$, and $\mathcal{A}$, proving judgments of the form $\mathcal{A}\langle \mathbf{m}, \overline{K_i}\rangle \nwarrow \mathbf{n}$, where $\overline{K_i}$ is a continuation stack. We call it the *Defunctionalized CPS (D-CPS) semantics* (Figure 3.2). The equivalence of the two semantics may be defined as follows: $(e \Downarrow \mathbf{n})$ iff $\mathcal{E}\langle e, \varepsilon\rangle \nwarrow \mathbf{n}$, and is easy to show by simple induction. We call the transformation from the natural semantics to the D-CPS semantics the *D-CPS transformation*.

$$\frac{\mathcal{A}\langle \mathbf{n}, \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{E}\langle \mathbf{n}, \overline{K_i}\rangle \nwarrow \mathbf{m}} \qquad \frac{\mathcal{E}\langle e, \mathbf{K1} : \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{E}\langle \mathbf{abs}\ e, \overline{K_i}\rangle \nwarrow \mathbf{m}} \qquad \frac{\mathcal{E}\langle e_1, \mathbf{K2}(e_2, \odot) : \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{E}\langle e_1 \odot e_2, \overline{K_i}\rangle \nwarrow \mathbf{m}}$$

$$\frac{\mathcal{A}\langle |\mathbf{n}|, \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{A}\langle \mathbf{n}, \mathbf{K1} : \overline{K_i}\rangle \nwarrow \mathbf{m}} \qquad \frac{\mathcal{E}\langle e_2, \mathbf{K3}(\mathbf{n_1}, \odot) : \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{A}\langle \mathbf{n_1}, \mathbf{K2}(e_2, \odot) : \overline{K_i}\rangle \nwarrow \mathbf{m}}$$

$$\frac{\mathcal{A}\langle \mathbf{n_1}\ [\odot]\ \mathbf{n_2}, \overline{K_i}\rangle \nwarrow \mathbf{m}}{\mathcal{A}\langle \mathbf{n_2}, \mathbf{K3}(\mathbf{n_1}, \odot) : \overline{K_i}\rangle \nwarrow \mathbf{m}} \qquad \mathcal{A}\langle \mathbf{m}, \varepsilon\rangle \nwarrow \mathbf{m}$$

Figure 3.2: A D-CPS semantics for arithmetic expressions

Note that the D-CPS semantics has a particular form: each rule has at most one premise, and the right-hand sides of the $\nwarrow$ symbol are identical for the premise and the conclusion. Thus, it is easy to transform the semantics into an abstract machine

(Figure 3.3), where the states are left-hand sides of the $\nwarrow$ symbol, each semantic rule with a premise is transformed into a transition rule for the machine (from the left-hand side of the conclusion to the left-hand side of the premise) and the rule with no premises becomes a halting state.

$$
\begin{array}{lcl}
\mathcal{E}\langle \mathbf{n}, \overline{K_i}\rangle & \Rightarrow & \mathcal{A}\langle \mathbf{n}, \overline{K_i}\rangle \\
\mathcal{E}\langle \mathbf{abs}\ e, \overline{K_i}\rangle & \Rightarrow & \mathcal{E}\langle e, \mathbf{K1} : \overline{K_i}\rangle \\
\mathcal{E}\langle e_1 \odot e_2, \overline{K_i}\rangle & \Rightarrow & \mathcal{E}\langle e_1, \mathbf{K2}(e_2, \odot) : \overline{K_i}\rangle \\
\mathcal{A}\langle \mathbf{m}, \varepsilon\rangle & \Rightarrow & \mathbf{m} \\
\mathcal{A}\langle \mathbf{n}, \mathbf{K1} : \overline{K_i}\rangle & \Rightarrow & \mathcal{A}\langle |\mathbf{n}|, \overline{K_i}\rangle \\
\mathcal{A}\langle \mathbf{n_1}, \mathbf{K2}(e_2, \odot) : \overline{K_i}\rangle & \Rightarrow & \mathcal{E}\langle e_2, \mathbf{K3}(\mathbf{n_1}, \odot) : \overline{K_i}\rangle \\
\mathcal{A}\langle \mathbf{n_2}, \mathbf{K3}(\mathbf{n_1}, \odot) : \overline{K_i}\rangle & \Rightarrow & \mathcal{A}\langle \mathbf{n_1}\ [\odot]\ \mathbf{n_2}, \overline{K_i}\rangle
\end{array}
$$

Figure 3.3: An abstract machine for arithmetic expressions

The equivalence of the D-CPS semantics and the abstract machine can be formulated as follows: $\mathcal{E}\langle e, \varepsilon\rangle \nwarrow \mathbf{n}$ iff $\mathcal{E}\langle e, \varepsilon\rangle \Rightarrow^* \mathbf{n}$, where $\Rightarrow^*$ is the reflexive and transitive closure of the relation $\Rightarrow$.

Though the transformation from the D-CPS semantics into the abstract machine-based semantics is trivial, the change is conceptually significant. The former is a big-step semantics, i.e., one that proves judgments on a relation between expressions, stacks and the final value. The latter is a small-step semantics, which describes separate steps of computation.

The D-CPS transformation is summaraized in Figure 3.4.

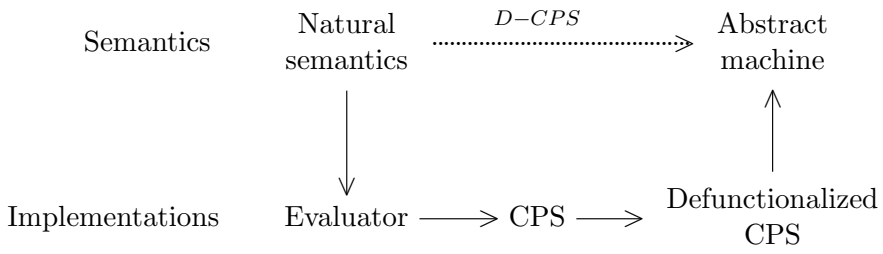


Figure 3.4: The D-CPS transformation.

3.2 Argument stack introduction

An essential flaw of the STG language natural semantics is its treatment of applications with too many arguments. Whenever an application lacks some arguments (the App1 rule is used) somewhere in the proof of the first premise of App3, there may be more arguments "waiting" in the second premise. Consider the following program (for arbitrary e and p):

$$\begin{array}{rl} \textbf{letrec} & f = \lambda_n\ x.e \\ & g = \lambda_n.f \\ \textbf{in} & g\ p \end{array}$$

First, in App3 g is evaluated in the first premise. g does not take any arguments, and then f is evaluated to itself by App1, because there are not enough arguments to proceed (the argument p is temporarily 'forgotten' during the computation of the 'argument bottleneck' g). Only then, in the second premise of the App3 rule, the expression $f\ p$ is created and evaluated.

To solve this problem, we introduce another entity to our judgments, which we call the *argument accumulator*. The judgments now take the form $\langle \Gamma, e, \overline{p_i} \rangle \Downarrow \langle \Delta, w, \overline{q_i} \rangle$, where $\overline{p_i}$ and $\overline{q_i}$ are the accumulators — stacks containing variables (intuitively, pointers). The intuition is that whenever we see an application, we put the arguments in the accumulator, and take them out when they are needed for entering a closure. The argument-accumulating semantics is given in Figure 3.5. The A-Accum rule is introduced. It deals with applications by putting arguments in the accumulator. All the other rules dealing with application are limited to applications to the empty tuple of arguments, while the 'real' arguments are now stored in the accumulator. The application rules App2 and App3 from the previous semantics are now merged to form the A-App2.5 rule. It is possible because the spare arguments do not need to be held back in the second premise of App3, but they travel up the proof in the accumulator and can be accessed when needed. Note that only constructors and applications to empty tuple of arguments are now normal forms.

The argument-accumulating semantics is sound and complete with respect to the STG language natural semantics.

Theorem 3 (soundness and completeness)**.** *If e is a closed expression, then:*

1. $(\emptyset : e \ \downarrow\ \Delta : C\ \overline{p_i})$ *iff* $\langle \emptyset, e, \varepsilon \rangle \Downarrow \langle \Delta, C\ \overline{p_i}, \varepsilon \rangle$,
2. $(\emptyset : e \ \downarrow\ \Delta : p\ \overline{p_i})$ *iff* $\langle \emptyset, e, \varepsilon \rangle \Downarrow \langle \Delta, p, \overline{p_i} \rangle$.

$$\langle \Gamma, C\ \overline{p_i}, \varepsilon\rangle \ \Downarrow\ \langle \Gamma, C\ \overline{p_i}, \varepsilon\rangle \qquad \text{A-Con}$$

$$\frac{\langle \Gamma, p, (p_1 \dots p_m\ q_1 \dots q_n)\rangle \ \Downarrow\ \langle \Delta, w, \overline{r_i}\rangle}{\langle \Gamma, (p\ p_1 \dots p_m), q_1 \dots q_n\rangle \ \Downarrow\ \langle \Delta, w, \overline{r_i}\rangle}\ m > 0 \qquad \text{A-Accum}$$

$$\langle \Gamma\{p \mapsto \lambda_n\ x_1 \dots x_m.e\}, p, (p_1 \dots p_n)\rangle \ \Downarrow\ \langle \Gamma, p, (p_1 \dots p_n)\rangle \ \ n < m \qquad \text{A-App1}$$

$$\frac{\langle \Gamma, e[x_1/p_1 \dots x_m/p_m], p_{m+1} \dots p_n\rangle \ \Downarrow\ \langle \Delta, w, \overline{r_i}\rangle}{\langle \Gamma\{p \mapsto \lambda_n\ x_1 \dots x_m.e\}, p, (p_1 \dots p_n)\rangle \ \Downarrow\ \langle \Delta, w, \overline{r_i}\rangle}\ m \le n \qquad \text{A-App2.5}$$

$$\frac{\langle \Gamma, e, \varepsilon\rangle \ \Downarrow\ \langle \Delta, C\ \overline{q_i}, \varepsilon\rangle}{\langle \Gamma\{p \mapsto \lambda_u.e\}, p, \varepsilon\rangle \ \Downarrow\ \langle \Delta \oplus [p \mapsto \lambda_n.C\ \overline{q_i}], C\ \overline{q_i}, \varepsilon\rangle} \qquad \text{A-App4}$$

$$\frac{\begin{array}{c}\langle \Gamma, e, \varepsilon\rangle \ \Downarrow\ \langle \Delta\{q \mapsto \lambda_n\ x_1 \dots x_k\ x_{k+1} \dots x_n.f\}, q, (q_1 \dots q_k)\rangle \\ \langle \Delta \oplus [p \mapsto \lambda_n\ x_{k+1} \dots x_n.f[x_1/q_1 \dots x_k/q_k]], \\ q, (q_1 \dots q_k\ p_1 \dots p_m)\rangle \ \Downarrow\ \langle \Theta, w, \overline{r_i}\rangle\end{array}}{\langle \Gamma\{p \mapsto \lambda_u.e\}, p, (p_1 \dots p_m)\rangle \ \Downarrow\ \langle \Theta, w, \overline{r_i}\rangle} \qquad \text{A-App5}$$

$$\frac{\langle \Gamma \oplus \overline{[p_i \mapsto lf_i[\overline{x_i/p_i}]]}, e[\overline{x_i/p_i}], \overline{r_i}\rangle \ \Downarrow\ \langle \Delta, w, \overline{s_i}\rangle}{\langle \Gamma, \mathbf{letrec}\ \overline{x_i = lf_i}\ \mathbf{in}\ e, \overline{r_i}\rangle \ \Downarrow\ \langle \Delta, w, \overline{s_i}\rangle}\ (*) \qquad \text{A-Letrec}$$

$$\frac{\langle \Gamma, e, \varepsilon\rangle \ \Downarrow\ \langle \Delta, C_k\ \overline{p_j}, \varepsilon\rangle \quad \langle \Delta, e_k[\overline{x_{kj}/p_j}], \overline{q_i}\rangle \ \Downarrow\ \langle \Theta, w, \overline{r_i}\rangle}{\langle \Gamma, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \overline{q_i}\rangle \ \Downarrow\ \langle \Theta, w, \overline{r_i}\rangle} \qquad \text{A-Case}$$

$$(*)\ \ \overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma)$$

Figure 3.5: The argument-accumulating semantics

3.3 Replacing substitution with environment

The next step toward an abstract machine is introduction of environments. This step is made simpler by the fact that in the argument-accumulating semantics for well-formed configurations we substitute only pointers for bound variables. Thus for each expression there will be an associated environment, which will bind addresses in the heap (pointers) with the free variables of the expression.

The *explicit environment semantics* is shown in Figure 3.6. It proves judgments of the form $\langle \Gamma, e, \sigma, \overline{q_i} \rangle \ \overline{\Downarrow} \ \langle \Delta, w, \tau, \overline{r_i} \rangle$, where σ and τ are environments, i.e., partial functions from variables to variables. We will denote the set of all environments by ENV. Γ, Δ and Θ are heaps of a new kind: their domain are variables from the set $\mathfrak{X}$ and their codomain are closures, i.e., elements of $lf \times ENV$. $FV(l)$ stands for the set of all free variables of the lambda form l. The environment $\sigma\restriction X$ is a subset of an environment σ with its domain trimmed to the set of variables X, $\sigma[\overline{x_i/p_i}]$ is an extension of σ by $[\overline{x_i/p_i}]$. We also write $e[\sigma]$ when we use the environment σ as a substitution. Intuitively, the argument accumulator stores pointers.

The trimming of environments is not essential for the soundness and completeness of the explicit environment semantics. We decided to leave the trimming in the E-Letrec rule and in the rules performing updates to indicate that closures in the heap are an abstraction of real-life closures in a real-life heap (where the closures contain only values for variables that are actually free in the function).

To avoid confusion, we will now denote heaps used in the argument-accumulating semantics by A-heap and heaps used in the explicit environment semantics by E-heap.

Definition 4. *1. An expression is* env-well-formed *iff the set of all its variables (both bound and free) is a subset of BOUND.*

2. An environment σ is env-well-formed *iff it is a function from BOUND to POINTERS.*

3. An expression e is closed by *an environment σ iff $FV(e) \subseteq Dom(\sigma)$.*

4. A E-heap Γ with $Dom(\Gamma) \subseteq POINTERS$ is env-well-formed *iff for each closure (e, σ) in Γ both e and σ are env-well-formed and e is closed by σ.*

The correspondence between an A-heap and a E-heap is defined as follows:

Definition 5. *An A-heap Γ and a E-heap $\Gamma^\bullet$ are* similar *iff:*

1. $Dom(\Gamma) = Dom(\Gamma^\bullet)$,

2. $\Gamma^\bullet$ is env-well-formed,

3. for any $p \in POINTERS$, if $\Gamma(p) = (\lambda_\nu \overline{y_i}.\widetilde{e})$ and $\Gamma^\bullet(p) = (\lambda_\mu \overline{x_i}.e, \tau)$ then $\overline{y_i} = \overline{x_i}$, $\widetilde{e} = e[\tau]$ and $\nu = \mu$.

$$\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon\rangle \ \overline{\Downarrow}\ \langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon\rangle \qquad \text{E-Con}$$

$$\frac{\langle \Gamma, x, \sigma, (\sigma x_1 \ldots \sigma x_m\ q_1 \ldots q_n)\rangle \ \overline{\Downarrow}\ \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\langle \Gamma, (x\ x_1 \ldots x_m), \sigma, q_1 \ldots q_n\rangle \ \overline{\Downarrow}\ \langle \Delta, w, \gamma, \overline{r_i}\rangle}\ m > 0 \qquad \text{E-Accum}$$

$$\langle \Gamma\{\sigma x \mapsto (\lambda_n\ x_1 \ldots x_m.e, \tau)\}, x, \sigma, p_1 \ldots p_n\rangle \ \overline{\Downarrow}\ \langle \Gamma, x, \sigma, p_1 \ldots p_n\rangle \ \ n < m \qquad \text{E-App1}$$

$$\frac{\langle \Gamma, e, \tau[x_1/p_1 \ldots x_m/p_m], p_{m+1} \ldots p_n\rangle \ \overline{\Downarrow}\ \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\langle \Gamma\{\sigma x \mapsto (\lambda_n\ x_1 \ldots x_m.e, \tau)\}, x, \sigma, p_1 \ldots p_n\rangle \ \overline{\Downarrow}\ \langle \Delta, w, \gamma, \overline{r_i}\rangle}\ m \leq n \qquad \text{E-App2.5}$$

$$\frac{\langle \Gamma, e, \tau, \varepsilon\rangle \ \overline{\Downarrow}\ \langle \Delta, C\ \overline{x_i}, \gamma, \varepsilon\rangle}{\langle \Gamma\{\sigma x \mapsto (\lambda_u.e, \tau)\}, x, \sigma, \varepsilon\rangle \ \overline{\Downarrow}\ \langle \Delta \oplus [\sigma x \mapsto (\lambda_n.C\ \overline{x_i}, \gamma\restriction\overline{x_i})], C\ \overline{x_i}, \gamma, \varepsilon\rangle} \qquad \text{E-App4}$$

$$\frac{\begin{array}{c}\langle \Gamma, e, \tau, \varepsilon\rangle \ \overline{\Downarrow}\ \langle \Delta\{\gamma y \mapsto (\lambda_n\ x_1 \ldots x_k\ x_{k+1} \ldots x_n.f, \mu)\}, y, \gamma, q_1 \ldots q_k\rangle \\ \langle \Delta \oplus [\sigma x \mapsto (\lambda_n\ x_{k+1} \ldots x_n.f, \mu[x_1/q_1 \ldots x_k/q_k])], \\ y, \gamma, q_1 \ldots q_k\ p_1 \ldots p_m\rangle \ \overline{\Downarrow}\ \langle \Theta, w, \xi, \overline{r_i}\rangle\end{array}}{\langle \Gamma\{\sigma x \mapsto (\lambda_u.e, \tau)\}, x, \sigma, p_1 \ldots p_m\rangle \ \overline{\Downarrow}\ \langle \Theta, w, \xi, \overline{r_i}\rangle} \qquad \text{E-App5}$$

$$\frac{\langle \Gamma \oplus \overline{[p_i \mapsto (lf_i, \tau_i[\overline{x_i/p_i}]\restriction \mathrm{FV}(lf_i))]}, e, \sigma[\overline{x_i/p_i}], \overline{r_i}\rangle \ \overline{\Downarrow}\ \langle \Delta, w, \gamma, \overline{s_i}\rangle}{\langle \Gamma, \textbf{letrec}\ \overline{x_i = lf_i}\ \textbf{in}\ e, \sigma, \overline{r_i}\rangle \ \overline{\Downarrow}\ \langle \Delta, w, \gamma, \overline{s_i}\rangle}\ (*) \qquad \text{E-Letrec}$$

$$\frac{\langle \Gamma, e, \sigma, \varepsilon\rangle \ \overline{\Downarrow}\ \langle \Delta, C_k\ \overline{y_j}, \gamma, \varepsilon\rangle \quad \langle \Delta, e_k, \sigma[\overline{x_{kj}/\gamma y_j}], \overline{q_i}\rangle \ \overline{\Downarrow}\ \langle \Theta, w, \xi, \overline{r_i}\rangle}{\langle \Gamma, \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}\rangle \ \overline{\Downarrow}\ \langle \Theta, w, \xi, \overline{r_i}\rangle} \qquad \text{E-Case}$$

$$(*)\ \ \overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma)$$

Figure 3.6: The explicit environment semantics

By $\Gamma^\bullet$ we will denote a E-heap that is similar to an A-heap Γ.

Theorem 6 (soundness and completeness)**.** *If*
e is a closed expression, then:

1. *If* $\langle \emptyset, e, \varepsilon \rangle \Downarrow \langle \Delta, \widetilde{w}, \overline{q_i} \rangle$ *then there exist* $\Delta^\bullet, w, \tau$ *s.t.* $\langle \emptyset, e, \emptyset, \varepsilon \rangle \overline{\Downarrow} \langle \Delta^\bullet, w, \tau, \overline{q_i} \rangle$ *and* $\widetilde{w} = w[\tau]$.

2. *If* $\langle \emptyset, e, \emptyset, \varepsilon \rangle \overline{\Downarrow} \langle \Delta^\bullet, w, \tau, \overline{q_i} \rangle$ *then there exists* Δ *s.t.* $\langle \emptyset, e, \varepsilon \rangle \Downarrow \langle \Delta, w[\tau], \overline{q_i} \rangle$.

3.4 Transformation to Defunctionalized CPS

We are now ready to perform the D-CPS transformation. It may be done in exactly the same manner as described in Section 3.1, and its result is shown in Figure 3.7. We call the resulting semantics the D-CPS semantics.

The rules E-App4 and E-App5 give rise to two continuations, but – since rules for them are the same – we may merge them into a single continuation **UPD** (for 'update'). The continuation for the E-Case is named **ALT** (for 'alternatives').

Theorem 7 (soundness and completeness)**.** *For any* Γ*,* e*,* σ *and* $\overline{p_i}$*, the following holds:* $\langle \Gamma, e, \sigma, \overline{p_i} \rangle \overline{\Downarrow} \langle \Delta, f, \gamma, \overline{q_i} \rangle$ *iff* $\mathcal{E}\langle \Gamma, e, \sigma, \overline{p_i}, \varepsilon \rangle \nwarrow \langle \Delta, f, \gamma, \overline{q_i} \rangle$.

3.5 From the D-CPS semantics to the abstract machine

The extraction of an abstract machine from the D-CPS semantics may be done exactly as described in Section 3.1. The resulting D-CPS machine is shown in Figure 3.8. The soundness and completeness is trivial and may be formulated as follows:

Theorem 8 (soundness and completeness)**.** *For any* Γ*,* e*,* σ *and* $\overline{p_i}$*,* $\mathcal{E}\langle \Gamma, e, \sigma, \overline{p_i}, \varepsilon \rangle \nwarrow \langle \Delta, f, \gamma, \overline{q_i} \rangle$ *iff* $\mathcal{E}\langle \Gamma, e, \sigma, \overline{p_i}, \varepsilon \rangle \overset{dcps}{\Longrightarrow}^* \langle \Delta, f, \gamma, \overline{q_i} \rangle$.

3.6 The STG machine

In this section we show that the D-CPS machine is in fact the Spineless Tagless G-machine in disguise and we compare the resulting machine to the original formulation by Peyton Jones and Salkild.

$$\mathcal{A}\langle \Gamma, w, \sigma, \overline{p_i}, \varepsilon\rangle \;\nwarrow\; \langle \Gamma, w\ \sigma\ , \overline{p_i}\rangle \qquad \text{D-Halt}$$

$$\frac{\mathcal{A}\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\mathcal{E}\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle} \qquad \text{D-Con}$$

$$\frac{\mathcal{E}\langle \Gamma, x, \sigma, (\sigma x_1 \ldots \sigma x_m\ q_1 \ldots q_n), \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\mathcal{E}\langle \Gamma, (x\ x_1 \ldots x_m), \sigma, q_1 \ldots q_n, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}\ m > 0 \qquad \text{D-Accum}$$

$$\frac{\mathcal{A}\langle \Gamma, x, \sigma, p_1 \ldots p_n, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_n\ x_1 \ldots x_m.e, \tau)\}, x, \sigma, p_1 \ldots p_n, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}\ n < m \qquad \text{D-App1}$$

$$\frac{\mathcal{E}\langle \Gamma, e, \tau[x_1/p_1 \ldots x_m/p_m], p_{m+1} \ldots p_n, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}{\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_n\ x_1 \ldots x_m.e, \tau)\}, x, \sigma, p_1 \ldots p_n, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{r_i}\rangle}\ m \leq n \qquad \text{D-App2.5}$$

$$\frac{\mathcal{E}\langle \Delta \oplus [p \mapsto (\lambda_n.C\ \overline{x_i}, \gamma \restriction \overline{x_i})], C\ \overline{x_i}, \gamma, \varepsilon, \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w\ \xi, \overline{s_i}\rangle}{\mathcal{A}\langle \Delta, C\ \overline{x_i}, \gamma, \varepsilon, \mathbf{UPD}(p, \varepsilon) : \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w\ \xi, \overline{s_i}\rangle} \qquad \text{D}\mathcal{A}\text{-App4}$$

$$\frac{\mathcal{E}\langle \Gamma, e, \tau, \varepsilon, \mathbf{UPD}(\sigma x, \overline{r_i}) : \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{q_i}\rangle}{\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_u.e, \tau)\}, x, \sigma, \overline{r_i}, \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{q_i}\rangle} \qquad \text{D}\mathcal{E}\text{-App4.5}$$

$$\frac{\begin{array}{c}\mathcal{E}\langle \Delta \oplus [p \mapsto (\lambda_n\ x_{k+1} \ldots x_n.f, \mu[x_1/q_1 \ldots x_k/q_k])],\\ y, \gamma, q_1 \ldots q_k\ p_1 \ldots p_m, \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle\end{array}}{\begin{array}{c}\mathcal{A}\langle \Delta\{\gamma y \mapsto (\lambda_n\ x_1 \ldots x_k\ x_{k+1} \ldots x_n.f, \mu)\},\\ y, \gamma, q_1 \ldots q_k, \mathbf{UPD}(p, p_1 \ldots p_m) : \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle\end{array}} \qquad \text{D}\mathcal{A}\text{-App5}$$

$$\frac{\mathcal{E}\langle \Gamma \oplus \overline{[p_i \mapsto (lf_i, \tau_i[\overline{x_i/p_i}] \restriction \mathrm{FV}(lf_i))]}, e, \sigma[\overline{x_i/p_i}], \overline{r_i}\ \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{s_i}\rangle}{\mathcal{E}\langle \Gamma, \mathbf{letrec}\ \overline{x_i = lf_i}\ \mathbf{in}\ e, \sigma, \overline{r_i}, \overline{S_i}\rangle \;\nwarrow\; \langle \Delta, w, \gamma, \overline{s_i}\rangle}\ (*) \qquad \text{D-Letrec}$$

$$\frac{\mathcal{E}\langle \Gamma, e, \sigma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle}{\mathcal{E}\langle \Gamma, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}, \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle} \qquad \text{D}\mathcal{E}\text{-Case}$$

$$\frac{\mathcal{E}\langle \Delta, e_k, \sigma[\overline{x_{kj}/\gamma y_j}], \overline{q_i}, \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle}{\mathcal{A}\langle \Delta, C_k\ \overline{y_j}\ \gamma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \;\nwarrow\; \langle \Theta, w, \xi, \overline{r_i}\rangle} \qquad \text{D}\mathcal{A}\text{-Case}$$

$$(*)\ \ \overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma)$$

Figure 3.7: The D-CPS semantics

3.6.1 Merging and splitting of rules

First, we notice that Q$\mathcal{A}$-App4 is of the form

$$\ldots \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \ldots C\ \overline{x_i} \ldots \rangle$$

and Q-Con is the only rule of the form

$$\mathcal{E}\langle \ldots C\ \overline{x_i} \ldots \rangle \overset{dcps}{\Longrightarrow} \ldots$$

Therefore we can replace Q$\mathcal{A}$-App4 with the following:

$$\begin{array}{c} \mathcal{A}\langle \Delta, C\ \overline{x_i}, \gamma, \varepsilon, \mathbf{UPD}(p, \varepsilon) : \overline{S_i} \rangle \\ \overset{dcps}{\Longrightarrow}\ \mathcal{A}\langle \Delta \oplus [p \mapsto \lambda_n . C\ \overline{x_i}\ (\gamma \lfloor \overline{x_i} \rfloor)], C\ \overline{x_i}, \gamma, \varepsilon, \overline{S_i} \rangle \end{array}$$

We can also split the Halt rule into two rules, one for each kind of normal forms:

$$\begin{array}{rcll} \mathcal{A}\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \varepsilon \rangle & \overset{dcps}{\Longrightarrow} & \langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon \rangle & \text{Q-Halt-Con} \\ \mathcal{A}\langle \Gamma, x, \sigma, \overline{p_i}, \varepsilon \rangle & \overset{dcps}{\Longrightarrow} & \langle \Gamma, x, \sigma, \overline{p_i} \rangle & \text{Q-Halt-App} \end{array}$$

The expression on the left-hand side of the rule Halt-App is an application with zero arguments (x), since the only rule of the form $\ldots \overset{dcps}{\Longrightarrow} \mathcal{A}\langle \ldots w \ldots \rangle$ where w is an application is Q-App1, in which w has no arguments.

We will now merge the Q-App1 rule with Q-Halt-App and, separately, with Q-App5. We replace these three rules with the following two:

$$\begin{array}{c} \mathcal{E}\langle \Gamma\{\sigma x \mapsto \lambda_n\ x_1 \ldots x_m . e\ \tau\}, x, \sigma, p_1 \ldots p_n, \varepsilon \rangle \\ \overset{dcps}{\Longrightarrow}\ \langle \Gamma, x, \sigma, p_1 \ldots p_n \rangle \quad \text{where } n < m, \end{array}$$

$$\begin{array}{c} \mathcal{E}\langle \Delta\{\gamma y \mapsto \lambda_n\ x_1 \ldots x_k\ x_{k+1} \ldots x_n . f\ \mu\}, y, \gamma, q_1 \ldots q_k \\ \mathbf{UPD}(p, p_1 \ldots p_n) : \overline{S_i} \rangle \quad \text{where } k < n \\ \overset{dcps}{\Longrightarrow}\ \mathcal{E}\langle \Delta \oplus [p \mapsto \lambda_n\ x_{k+1} \ldots x_n . f\ \mu[x_1/q_1 \ldots x_k/q_k]], \\ y, \gamma, q_1 \ldots q_k\ p_1 \ldots p_n, \overline{S_i} \rangle. \end{array}$$

3.6.2 Introduction of the STG instructions

So far we have used two kinds of 'instructions': *eval* ($\mathcal{E}$) and *apply* ($\mathcal{A}$), where $\mathcal{E}$ intuitively means that we are currently evaluating an expression, and $\mathcal{A}$ means that we have just finished evaluating an expression and we need an element from the stack of continuations to go on.

After the merging of rules, we notice that the $\mathcal{A}$ instruction applies now only to the rules for constructors. We will dub such rules **return**.

$$\mathcal{A}\langle \Gamma, w, \sigma, \overline{p_i}, \varepsilon\rangle \overset{dcps}{\Longrightarrow} \langle \Gamma, w, \sigma, \overline{p_i}\rangle \qquad \text{Q-Halt}$$

$$\mathcal{E}\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{A}\langle \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \qquad \text{Q-Con}$$

$$\mathcal{E}\langle \Gamma, (x\ x_1 \ldots x_m), \sigma, q_1 \ldots q_n, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Gamma, x, \sigma, (\sigma x_1 \ldots \sigma x_m\ q_1 \ldots q_n), \overline{S_i}\rangle \quad \text{where } m > 0 \qquad \text{Q-Accum}$$

$$\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_n\ x_1 \ldots x_m.e, \tau)\}, x, \sigma, p_1 \ldots p_n, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{A}\langle \Gamma, x, \sigma, p_1 \ldots p_n, \overline{S_i}\rangle \quad n < m \qquad \text{Q-App1}$$

$$\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_n\ x_1 \ldots x_m.e, \tau)\}, x, \sigma, p_1 \ldots p_n, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Gamma, e, \tau[x_1/p_1 \ldots x_m/p_m], p_{m+1} \ldots p_n, \overline{S_i}\rangle \quad m \leq n \qquad \text{Q-App2.5}$$

$$\mathcal{A}\langle \Delta, C\ \overline{x_i}, \gamma, \varepsilon, \mathbf{UPD}(p, \varepsilon) : \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Delta \oplus [p \mapsto (\lambda_n.C\ \overline{x_i}, \gamma\restriction\overline{x_i})], C\ \overline{x_i}, \gamma, \varepsilon, \overline{S_i}\rangle \qquad \text{Q}\mathcal{A}\text{-App4}$$

$$\mathcal{E}\langle \Gamma\{\sigma x \mapsto (\lambda_u.e, \tau)\}, x, \sigma, \overline{r_i}, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Gamma, e, \tau, \varepsilon, \mathbf{UPD}(\sigma x, \overline{r_i}) : \overline{S_i}\rangle \qquad \text{Q}\mathcal{E}\text{-App4.5}$$

$$\mathcal{A}\langle \Delta\{\gamma y \mapsto (\lambda_n\ x_1 \ldots x_k\ x_{k+1} \ldots x_n.f, \mu)\}, y, \gamma, q_1 \ldots q_k, \mathbf{UPD}(p, p_1 \ldots p_m) : \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Delta \oplus [p \mapsto (\lambda_n\ x_{k+1} \ldots x_n.f, \mu[x_1/q_1 \ldots x_k/q_k])], y, \gamma, (q_1 \ldots q_k\ p_1 \ldots p_m), \overline{S_i}\rangle \qquad \text{Q}\mathcal{A}\text{-App5}$$

$$\mathcal{E}\langle \Gamma, \mathbf{letrec}\ \overline{x_i = lf_i}\ \mathbf{in}\ e, \sigma, \overline{r_i}, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Gamma \oplus \overline{[p_i \mapsto (lf_i, \tau_i[\overline{x_i/p_i}]\restriction \mathrm{FV}(lf_i))]}, e, \sigma[\overline{x_i/p_i}], \overline{r_i}, \overline{S_i}\rangle \quad (*) \qquad \text{Q-Letrec}$$

$$\mathcal{E}\langle \Gamma, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}, \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Gamma, e, \sigma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \qquad \text{Q}\mathcal{E}\text{-Case}$$

$$\mathcal{A}\langle \Delta, C_k\ \overline{y_j}, \gamma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \overset{dcps}{\Longrightarrow} \mathcal{E}\langle \Delta, e_k, \sigma[\overline{x_{kj}/\gamma y_j}], \overline{q_i}, \overline{S_i}\rangle \qquad \text{Q}\mathcal{A}\text{-Case}$$

$(*)$ $\overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma)$

Figure 3.8: The D-CPS abstract machine

We also split the $\mathcal{E}$ instruction into two: one for application to zero arguments (we will dub such rules **enter**), and for any other kind of expression (dubbed **eval**).

We notice that now there is no rule for configurations of the form $\langle \mathbf{eval}, \Gamma, x, \ldots \rangle$, where x is a single variable, therefore we abandon the side condition $m > 0$ in the Q-Accum rule, so that evaluating an application to zero arguments means entering the closure it represents.

The changes in the machine are summarized in Figure 3.9. We claim that this machine is the STG machine up to some minor details described in the following subsection. As evidence, in Figure 3.9 we put numbers next to names of the rules; these numbers are the numbers of the transition rules in Peyton Jones and Salkild's original STG machine [33] (not all numbers are present since our machine lacks primitive arithmetics and default alternatives in **case** expressions, and Halt rules are not featured in the original STG machine).

3.6.3 Soundness and completeness

We can combine all the local soundness and completeness theorems to formulate our main proposition. Recall that by Δ and $\Delta^\bullet$ we denote a pair of similar heaps (Definition 5).

Theorem 9 (completeness)**.** *For a closed expression e, the following hold:*

1. *If $(\emptyset : e \downarrow \Delta : p\ \overline{p_i})$, there exist $\Delta^\bullet$, x and σ such that $\langle \mathbf{eval}, \emptyset, e, \varepsilon, \varepsilon, \varepsilon \rangle \overset{stg}{\Longrightarrow}^* \langle \Delta^\bullet, x, \sigma, \overline{p_i} \rangle$ and $\sigma x = p$.*

2. *If $(\emptyset : e \downarrow \Delta : C\ \overline{p_i})$, there exist $\Delta^\bullet$, $\overline{x_i}$ and σ such that: $\langle \mathbf{eval}, \emptyset, e, \varepsilon, \varepsilon, \varepsilon \rangle \overset{stg}{\Longrightarrow}^* \langle \Delta^\bullet, C\ \overline{x_i}, \sigma, \varepsilon \rangle$ and $\sigma \overline{x_i} = \overline{p_i}$.*

Theorem 10 (soundness)**.** *For a closed expression e, the following hold:*

1. *If $\langle \mathbf{eval}, \emptyset, e, \varepsilon, \varepsilon, \varepsilon \rangle \overset{stg}{\Longrightarrow}^* \langle \Delta^\bullet, x, \sigma, \overline{p_i} \rangle$ then there exists Δ such that $(\emptyset : e \downarrow \Delta : (\sigma x)\ \overline{p_i})$.*

2. *If $\langle \mathbf{eval}, \emptyset, e, \varepsilon, \varepsilon, \varepsilon \rangle \overset{stg}{\Longrightarrow}^* \langle \Delta^\bullet, C\ \overline{p_i}, \sigma, \varepsilon \rangle$ then there exists Δ such that $(\emptyset : e \downarrow \Delta : C\ \overline{p_i})$.*

3.6.4 Design differences

The original STG machine was designed, while ours was derived. Still, the design choices we have made when introducing successive semantics have a great impact on the final machine. In this section we compare the STG machine from Figure 3.9 with the machine described by Peyton Jones.

$$\langle \mathbf{return}, \Gamma, C\ \overline{x_i}, \sigma, \overline{p_i}, \varepsilon\rangle \overset{stg}{\Longrightarrow} \langle \Gamma, C\ \overline{x_i}, \sigma, \overline{p_i}\rangle \qquad \text{S-Halt-Con}$$

$$\langle \mathbf{enter}, \Gamma\{\sigma x \mapsto (\lambda_n\ x_1 \dots x_m.e, \tau)\}, x, \sigma, p_1 \dots p_n, \varepsilon\rangle \overset{stg}{\Longrightarrow} \langle \Gamma, x, \sigma, p_1 \dots p_n\rangle \quad n < m \qquad \text{S-App1Halt}$$

$$\langle \mathbf{eval}, \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{return}, \Gamma, C\ \overline{x_i}, \sigma, \varepsilon, \overline{S_i}\rangle \qquad \text{S-Con (5)}$$

$$\langle \mathbf{eval}, \Gamma, (x\ x_1 \dots x_m), \sigma, q_1 \dots q_n, \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{enter}, \Gamma, x, \sigma, (\sigma x_1 \dots \sigma x_m\ q_1 \dots q_n), \overline{S_i}\rangle \qquad \text{S-Accum (1)}$$

$$\langle \mathbf{enter}, \Delta\{\gamma y \mapsto (\lambda_n\ x_1 \dots x_k\ x_{k+1} \dots x_n.f, \mu)\}, y, \gamma, q_1 \dots q_k, \mathbf{UPD}(p, p_1 \dots p_m) : \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{enter}, \Delta \oplus [p \mapsto (\lambda_n\ x_{k+1} \dots x_n.f, \mu[x_1/q_1 \dots x_k/q_k])], y, \gamma, (q_1 \dots q_k\ p_1 \dots p_m), \overline{S_i}\rangle \quad k < n \qquad \text{S-App1App5 (17)}$$

$$\langle \mathbf{enter}, \Gamma\{\sigma x \mapsto (\lambda_n\ x_1 \dots x_m.e, \tau)\}, x, \sigma, p_1 \dots p_n, \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{eval}, \Gamma, e, \tau[x_1/p_1 \dots x_m/p_m], p_{m+1} \dots p_n, \overline{S_i}\rangle \quad m \leq n \qquad \text{S-App2.5 (2)}$$

$$\langle \mathbf{return}, \Delta, C\ \overline{x_i}, \gamma, \varepsilon, \mathbf{UPD}(p, \varepsilon) : \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{return}, \Delta \oplus [p \mapsto (\lambda_n.C\ \overline{x_i}, \gamma\restriction\overline{x_i})], C\ \overline{x_i}, \gamma, \varepsilon, \overline{S_i}\rangle \qquad \text{S}\mathcal{A}\text{-App4Con (16)}$$

$$\langle \mathbf{enter}, \Gamma\{\sigma x \mapsto (\lambda_u.e, \tau)\}, x, \sigma, \overline{r_i}, \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{eval}, \Gamma, e, \tau, \varepsilon, \mathbf{UPD}(\sigma x, \overline{r_i}) : \overline{S_i}\rangle \qquad \text{S}\mathcal{E}\text{-App4.5 (15)}$$

$$\langle \mathbf{eval}, \Gamma, \mathbf{letrec}\ \overline{x_i = lf_i}\ \mathbf{in}\ e, \sigma, \overline{r_i}, \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{eval}, \Gamma \oplus [\overline{p_i \mapsto (lf_i, \tau_i[\overline{x_i/p_i}]\restriction \mathrm{FV}(lf_i))}], e, \sigma[\overline{x_i/p_i}], \overline{r_i}, \overline{S_i}\rangle \quad (*) \qquad \text{S-Letrec (3)}$$

$$\langle \mathbf{eval}, \Gamma, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}, \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{eval}, \Gamma, e, \sigma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \qquad \text{S}\mathcal{E}\text{-Case (4)}$$

$$\langle \mathbf{return}, \Delta, C_k\ \overline{y_j}, \gamma, \varepsilon, \mathbf{ALT}(\overline{C_i\ \overline{x_{ij}} \to e_i}, \sigma, \overline{q_i}) : \overline{S_i}\rangle \overset{stg}{\Longrightarrow} \langle \mathbf{eval}, \Delta, e_k, \sigma[\overline{x_{kj}/\gamma y_j}], \overline{q_i}, \overline{S_i}\rangle \qquad \text{S}\mathcal{A}\text{-Case (6)}$$

$$(*)\ \ \overline{p_i} \in \mathit{POINTERS} \setminus \mathit{Dom}(\Gamma)$$

Figure 3.9: The STG machine

Stacks Our formulation of the STG machine has two stacks (argument accumulator and continuation stack), while the original STG machine has three stacks (argument stack, return stack, and update stack). works exactly like the argument stack of the original STG machine, while the continuation stack covers the return stack and the update stack. Our two-stack machine can be simulated by a single-stack machine in exactly the same way as the original STG machine [33], since the harmonics of the stack operations in both machines are identical. However, we find the formulation with two stacks particularly clean (as opposed to single stack), since the return-update stack may be seen as a continuation (in particular, when we *finish* evaluation, we *continue* with an update), while the argument stack may not (we do not use arguments with computed normal forms, instead we constantly shuffle the argument stack during evaluation).

Instructions and environments The **enter** and **return** instructions are formulated slightly differently: here, **enter** takes a variable x and an environment σ, and then enters the closure under the address σx, while the original **enter** rule takes the address. Similarly, **return** expects a constructor expression (a constructor name and a tuple of variables) and an environment, while in the original formulation it needs a constructor name and a tuple of addresses. The equivalence of both approaches is almost trivial.

Problems with ill-typed expressions As pointed out by Encina and Peña [13], the original STG machine might not behave as expected for some ill-formed programs. For example, consider the following program (which would be ill-typed in any reasonable strongly-typed language):

$$\begin{array}{rl} \textbf{letrec} & id = \lambda_n\, x.x \\ & c = \lambda_u.C \\ & f = \lambda_u.\textbf{case}\ id\ \textbf{of}\ C \to D \\ \textbf{in} & f\ c \end{array}$$

The original machine allocates the declarations, pushes the pointer to c on the argument stack and continues with evaluation of f. The **case** expression in f first computes id, which evaluates to C (we have an argument for it on the argument stack!). The whole expression finally evaluates to D.

Even though the STG language is not typed, the intuition is that the evaluation should be broken in the **case** expression, since id should not get its argument. Indeed, it is the case when the three stacks of the original STG machine are simulated by a single stack, where the argument is "guarded" by an element containing **case** alternatives. This is a minor flaw, since for well-typed programs the machine with one stack behaves exactly like the machine with three stacks.

Our formulation avoids this problem by storing the argument stack in the continuation when evaluating the inner expression in the D$\mathcal{E}$-Case rule. The original STG machine

behavior would be obtained if we leave the argument stack for the first premise in the argument-accumulating semantics in the A-Case rule:

$$\frac{\langle \Gamma, e, \overline{q_i} \rangle \;\Downarrow\; \langle \Delta, C_k\ \overline{p_j}, \overline{r_i} \rangle \quad \langle \Delta, e_k[\overline{x_{kj}/p_j}], \overline{r_i} \rangle \;\Downarrow\; \langle \Theta, w, \overline{s_i} \rangle}{\langle \Gamma, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \overline{q_i} \rangle \;\Downarrow\; \langle \Theta, w, \overline{s_i} \rangle}$$

An alternative update rule The App5 rule from our natural semantics (and its successors: A-App5, E-App5 and S-App1App5) are problematic to implement in a compiler since we cannot create new expressions on the fly. Moreover, updating the heap as follows:

$$\Delta \oplus [p \mapsto (\lambda_n\ x_{k+1} \ldots x_n.f, \mu[x_1/q_1 \ldots x_k/q_k])]$$

would in practice mean modifying the expressions because environments are precomputed, i.e., each variable in an expression is statically bound to the n-th element on the stack, the n-th slot of the current closure, or a register. One solution is to update the heap not with the partially applied lambda-form, but with the computed normal form:

$$\frac{\begin{array}{c}\Gamma : e \;\downarrow\; \Delta\{q \mapsto \lambda_n\ x_1 \ldots x_k\ x_{k+1} \ldots x_n.f\} : q\ q_1 \ldots q_k \\ \Delta \oplus [p \mapsto \lambda_n.\ q\ q_1 \ldots q_k] : q\ q_1 \ldots q_k\ p_1 \ldots p_n \;\downarrow\; \Theta : w\end{array}}{\Gamma\{p \mapsto \lambda_u.e\} : p\ p_1 \ldots p_n \;\downarrow\; \Theta : w}$$

Now it is sufficient to create code for partial applications for all possible number of arguments. This approach is used in the original formulation of the STG machine as an alternative update rule. [1]

3.7 Extensions

An advantage of using a constructive derivation method for obtaining an abstract machine from the underlying natural semantics is its scalability. Any changes in the latter smoothly ensue in the machine.

A useful example of such an operation is adding primitive (unboxed) arithmetics. We need to include numerical literals and operators as base constructions in the language, and to extend the STG natural semantics with a few intuitive rules, with no need to alter any of the original rules. If careful when introducing environments, we can augment the machine with a primitive arithmetics similar to the one presented in the original papers [33, 35].

In the same way we can add other concepts that are easy to express in the natural semantics, but may not be that trivial in the machine, like (monadic) input/output, or the Haskell *seq* operator.

[1] The original STG rule to which we would come via all our transformations is called 17a.

3.8 Formalization in Coq

The semantics presented in this chapter are formalized in a similar way to the STG natural semantics.

In semantics which use environments, while adding new elements while going past n binders, we need to shift the old environment and paste the newly bound elements (see Section 2.3.3):

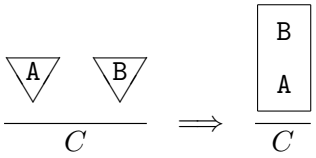
```
Definition shift_assoc (offset : nat) (na : nat * var) :
  nat * var :=
match na with
| (n, a) ⇒ (n + offset, a)
end.

Definition shift (offset : nat) (xs : env) : env :=
map (shift_assoc offset) xs.

(...)
| E_App2_5 : forall Gamma Delta m e x tau p pn w rs sigma rho,
  env_find sigma x = Some (Atom p) →
  Gamma p = Some (Lf_n m e, tau) →
  m <= length pn →
  ($ Gamma $ e $ zip_var_list m (firstn m pn) ++ shift m tau $
           skipn m pn ⇊ Delta $ w $ rho $ rs) →
  ($ Gamma $ App x nil $ sigma $
           pn ⇊ Delta $ w $ rho $ rs)
(...)
```

Proving soundness and completeness

All the proofs of soundness and completeness are done either by a standard structural induction on derivations, or on the height of proofs. The latter is specially useful when proving soundness of semantics which introduce stacks (argument-accumulating and D-CPS), where some rules undergoes a transformation of the shape:

$$\frac{a \downarrow v_1 \quad b(v_1) \downarrow v_2}{c \downarrow v_2} \implies \frac{\langle a, b(\cdot)::s\rangle \downarrow v_2}{\langle c, s\rangle \downarrow v_2} + \frac{\langle b(v_1), s\rangle \downarrow v_2}{\langle v_1, b(\cdot)::s\rangle \downarrow v_2}$$

The tree-like proof in the first semantics becomes more list-like:



Since Coq uses a constructive logic, to prove soundness, we need to transform the list-like proof $\begin{smallmatrix}\texttt{B}\\ \texttt{A}\end{smallmatrix}$ into two separate proofs, `A` and `B`. The problem is with the part of the list-like proof which corresponds to the proof `A`, because it is not a structural subterm, and thus it is impossible to get a standard Coq inductive assumption. To solve the problem, we use a well-founded induction on heights of the derivations (`lt_wf_ind`). To measure the height, for each list-like semantics we introduce a twin with additional element of judgments, the height (zero for axioms, successor of maximum for premises for non-axiom rules).

Correctness of the Spineless Tagless G-machine

A single step of the STG machine is formalized as a relation between subsequent states:

```
Inductive STG : instruction → heapB → expr → env → vars → stack
              → instruction → heapB → expr → env → vars → stack
              → Prop := (...)
```

The whole execution is described by a subset of the transitive closure of the `STG` relation which ends with a halting state:

```
Inductive STG_run : instruction
                  → heapB → expr → env → vars → stack
                  → heapB → expr → env → vars
                  → Prop := (...)

Notation "($ a $ b $ c $ d $ e $ f ⟹* g $ h $ i $ j )" :=
  (STG_run a b c d e f g h i j).
```

To justify the formulation of the completeness theorem, we prove that only application and constructors are values:

```
Theorem values :
forall Gamma e Delta f,
  ($ Gamma $ e ↓ Delta $ f) →
(exists x, exists xs, f = App x xs) ∨
(exists C, exists xs, f = Constr C xs).
Proof. (...) Qed.
```

The completeness theorem may be stated as follows:

```
Theorem completeness :
forall e x xs C Delta,
    closed_expr 0 e →
    no_atoms e →
(($ emptyA $ e ↓ Delta $ App x xs) →
    exists DeltaP, exists y, exists sigma,
       ($ Eval $ emptyB $ e $ nil $ nil $ nil ⟹*
         DeltaP $ App y nil $ sigma $ xs) ∧
       similar Delta DeltaP ∧
       App y nil ~ [sigma] = App x nil)
∧
(($ emptyA $ e ↓ Delta $ Constr C xs) →
    exists DeltaP, exists ys, exists sigma,
       ($ Eval $ emptyB $ e $ nil $ nil $ nil ⟹*
         DeltaP $ Constr C ys $ sigma $ nil) ∧
       similar Delta DeltaP ∧
       Constr C ys ~ [sigma] = Constr C xs).
Proof. (...) Qed.
```

Since arguments of partial applications are held in the stack, we need an operation to join it to the expression:

```
Fixpoint join_expr (Ds : list atom) (e : expr) {struct e} :
  expr :=
match e with
| App v vs ⇒ App v (vs ++ map Atom Ds)
| Constr c vs ⇒ Constr c vs
| Letrec dfs f ⇒ Letrec dfs (join_expr Ds f)
| Case e als ⇒ Case e (map (join_alt Ds) als)
end
with join_alt (Ds : list atom) (a : alt)
  {struct a} : alt :=
match a with
| Alt c b f ⇒ Alt c b (join_expr Ds f)
end.

Notation " e >< d " := (join_expr d e) (at level 70).
```

Now, we may formulate a soundness theorem as follows:

```
Theorem soundness :
forall (e f : expr) (DeltaP : heapB) (sigma : env)
       (Cs : list atom),
    closed_expr 0 e →
    no_atoms e →
    ($ Eval $ emptyB $ e $ nil $ nil $ nil ⇒*
      DeltaP $ f $ sigma $ map Atom Cs) →
exists Delta,
    similar Delta DeltaP ∧
    ($ emptyA $ e ↓ Delta $ (f ~[sigma] ⋈ Cs)).
Proof. (...) Qed.
```

3.9 Related work

The idea of deriving lazy machines from natural semantics was first proposed by Sestoft [39]. He used an informal method to change rules for constructing proofs in natural semantics into rules for constructing a sequence of machine states. Then Mountjoy suggested that the same method for an extended semantics may lead to a machine that is closer to STG, and gave a proof of equivalence of some more elaborate abstract machines (but still far from the STG machine) [31]. The work of Mountjoy was continued by Encina and Peña [13,12,14]. They used similar methods to invent STG-like machines and gave detailed proofs of their equivalence with an initial natural semantics.[2]

Though our approach may at first seem similar to the Encina and Peña's, it is based on different principles. To underscore the differences, we will examine the four main concepts: languages, semantics, derivations and abstract machines.

Languages In their articles, Encina and Peña introduce two new languages, both bearing the same name Fun. While neither of them is very different from STG, they were designed to fit the sole purpose of proving equivalence of a semantics and a machine.

Our approach, in turn, is to take the well-known STG language exactly as introduced by Peyton Jones, and give it a natural semantics, which is an interesting challenge even outside the context of deriving abstract machines. Nevertheless, starting with the natural semantics for *the* STG language was the key to obtaining *the* STG machine.

[2] In [14] Encina and Peña present two machines: push/enter and eval/apply. Obviously, we are interested only in the former.

Semantics In our work, semantics for **letrec** and **case** expressions are similar to Encina and Peña's. They follow the approach of Launchbury and Sestoft. The key difference is in the treatment of multiple λ-binders and partial applications.

The two semantics for both Fun languages consequently evaluate partial applications by allocation in the heap. They allocate either a primitive heap element *pap*, or the lambda-form with the actual arguments substituted by the corresponding prefix of formal arguments. Though in the machine this allocation may be fused with an update, we do not find such solution elegant when concerning natural semantics.

Encina and Peña admit that their semantics, just as their languages, are tailored for the transformation into a particular machine. Our ambition, on the other hand, is to propose a more general and intuitive natural semantics, ready for any other formal reasoning, like preservation of semantics by program transformations in optimizing compilers.

We are also the first to address update flags in the semantics, which, if assigned correctly by a static program analysis, lead to a boost of performance.

Derivations Encina and Peña present their machines, but they do not explain how they invented them. They only refer to Sestoft's approach, who used his intuition of flattening proof trees into sequences of machine transitions. This is hardly a derivation understood as a transformation from one entity (in this case a semantics) into another (an abstract machine) using a well-defined method. Moreover, their machines do not implement exactly the same evaluation model as their semantics (for example, S3 from [14] allocates more closures then the corresponding machine).

Our machine is a result of a method strongly inspired by a well-known transformation of programs, which preserves most important properties, including the evaluation model.

Abstract machines Both Fun languages are different than STG, thus their STG-like machines differ from the original STG machine. The most striking difference is the lack of **enter**, **eval** and **return** instructions which are STG-tuned incarnations of the eval ($\mathcal{E}$) and apply ($\mathcal{A}$) instructions arising naturally from the D-CPS transformation.

4 Certified compilation

A compiler is certified if it preserves the semantics of compiled expressions (up to a certain equivalence of final values), and it has been proven correct, preferably in an automatic proof assistant/checker like Coq, Isabelle/HOL, Twelf, etc. Certified compilers go hand in hand with certified software, which also requires a formal proof of correctness with respect to the semantics.

In this chapter we give a formal definition of certified compilation and describe some more usable variants. We also give an example of a certified compiler, which translates arithmetic expressions into reverse Polish notation. We also formalize the compiler and a proof of its correctness in Coq.

4.1 Certified compilation, formally

Let L_1, L_2 be languages, C_1, C_2 be sets called configurations, V_1, V_2 be sets called values, and $\Downarrow_1 \subseteq C_1 \times V_1$ and $\Downarrow_2 \subseteq C_2 \times V_2$ be relations called semantics. For two functions $init_1 : L_1 \to C_1$ and $init_2 : L_2 \to C_2$ called initial configurations and a relation $\sim \subseteq V_1 \times V_2$, a function $f : L_1 \to L_2$ is called a certified compiler iff the following has been proven (preferably in a proof system like Coq) for all $e \in L_1$ and $v_1 \in V_1$:

$$\text{If } init_1(e) \Downarrow_1 v_1 \text{ then there exists } v_2 \text{ such that } init_2(f(e)) \Downarrow_2 v_2 \text{ and } v_1 \sim v_2. \quad (4.1)$$

The relation $\sim$ is sometimes considered to be a function. In such case the previous condition simplifies to:

$$\text{If } init_1(e) \Downarrow_1 v_1 \text{ then } init_2(f(e)) \Downarrow_2 \sim(v_1). \quad (4.2)$$

The $\sim$ relation often satisfies some homomorphic properties. For example, consider a typical natural semantics for arithmetic expressions which evaluates to natural numbers ($\mathbb{N}$), and expressions in reverse Polish notation (interpreted as strings of symbols and denoted by RPN) which are evaluated by an abstract machine with a stack. The type of configurations of such machine is $RPN \times \mathbb{N}^*$. The machine finishes evaluation when reaching an empty expression and a singleton on the stack. In this setting, the $\sim$ relation may be defined as a function: $\sim(n) = (\varepsilon, [n])$. (The example of reverse Polish notation will be elaborated.)

We will also consider programs with input, understood as substitution for a free variable in a program. For example, a value x from a set $INPUT$ may be an input to a

program e, which we denote by $e\{x\}$. We assume that all compilers are invariant to input, that is $f(e\{x\}) = f(e)\{x\}$, for any compiler f.

A program e is *certified* if it is proven correct (preferably in a proof system like Coq) with respect to the $\Downarrow_1$ semantics. In other words:

$$\text{For all } x \in \mathit{INPUT} \text{ and } v_1 \in V_1, \text{ if } e\{x\} \Downarrow_1 v_1 \text{ then } \Theta(x, v_1), \tag{4.3}$$

where Θ describes what it means that a value is correct w.r.t. the input.

4.1.1 Soundness

Both conditions (4.1) and (4.2) are implications stating that if an expression has a value in the first semantics, it has an equivalent value in the second semantics, thus they state the correctness of the compiler. The question arises about the reverse (soundness), that is whether the following condition should be satisfied for all $e \in L_1$ and $v_2 \in V_2$:

$$\text{If } \mathit{init}_2(f(e)) \Downarrow_2 v_2 \text{ then there exists } v_1 \text{ such that } \mathit{init}_1(e) \Downarrow_1 v_1 \text{ and } v_1 \sim v_2. \tag{4.4}$$

To answer the question, we need to introduce some new notions:

1. A semantics $\Downarrow$ is *deterministic* if for all $c \in C$ and $v, v' \in V$, if $c \Downarrow v$ and $c \Downarrow v'$ then $v = v'$.

2. If the program always *terminates* in a sematics $\Downarrow$, it means that it has a value in $\Downarrow$ for any input.

If a semantics and programs satisfy both properties – which is a quite common scenario in applications – completeness implies soundness, hence condition (4.1) is sufficient. It is formulated in the following theorem.

Theorem 11. *For semantics $\Downarrow_1, \Downarrow_2$, if $\Downarrow_2$ is deterministic then for a certified compiler f, f is sound on programs that always terminate in $\Downarrow_1$.*

Proof. We show that property (4.4) holds for f. Let $\mathit{init}_2(f(e)) \Downarrow_2 v_2$ for some v_2 and e that terminates in $\Downarrow_1$. Since e terminates in $\Downarrow_1$, there exists v_1 s.t. $\mathit{init}_1(e) \Downarrow_1 v_1$, and so, because f is certified, v_2' s.t. $\mathit{init}_2(f(e)) \Downarrow_2 v_2'$ and $v_1 \sim v_2'$. The semantics $\Downarrow_2$ is deterministic, so $v_2' = v_2$, thus $v_1 \sim v_2$. □

Certification of implementations of recursively enumerable (semi-decidable) algorithms is much rarer (for example, all programs written in Coq terminate) and needs different definition of certification ($e\{x\} \Downarrow_1 v_1$ iff $\Theta(x, v_1)$). For a program e which does not terminate for an input x, there is no v_1 such that $\mathit{init}_1(e\{x\}) \Downarrow_1 v_1$, but, if the compiler is not sound, there may be some v_2 such that $\mathit{init}_2(f(e)\{x\}) \Downarrow_2 v_2$. Thus, after compilation, the program gives a wrong answer, though it should stuck in an infinite loop.

4.1.2 Partial compilers

In practice, compiler is often a partial function, because not all properly constructed terms of a language constitute correct programs. For example, we do not expect a compiler to compile an ill-typed program. In such case a definition of certified compiler may be formulated as:

$$\begin{array}{l}\text{For all } e \text{ and } v_1 \text{ s.t. } f(e) \text{ is defined, if } \mathit{init}_1(e) \Downarrow_1 v_1 \\ \text{then there exists } v_2 \text{ such that } \mathit{init}_2(f(e)) \Downarrow_2 v_2 \text{ and } v_1 \sim v_2.\end{array} \tag{4.5}$$

Obviously, if f is empty, that is it is not defined for any expression, it is a correct partial certified compiler. For this reason it would be desirable to guarantee that f is defined for a sufficiently wide class of programs. The perfect definition would be:

$$\begin{array}{l}\text{For all } e \text{ and } v_1, \text{ if } \mathit{init}_1(e) \Downarrow_1 v_1 \text{ then } f(e) \text{ is defined} \\ \text{and there exists } v_2 \text{ such that } \mathit{init}_2(f(e)) \Downarrow_2 v_2 \text{ and } v_1 \sim v_2.\end{array} \tag{4.6}$$

But proving this property for a real-life compiler seems almost impossible.

For some languages there are natural candidates for the class of compilable programs. For example, for a strongly-typed language, the following definition is natural and much easier to prove for a compiler:

$$\begin{array}{l}\text{For all } e \text{ and } v_1, \text{ if } e \text{ is well-typed and } \mathit{init}_1(e) \Downarrow_1 v_1 \text{ then } f(e) \text{ is defined} \\ \text{and there exists } v_2 \text{ such that } \mathit{init}_2(f(e)) \Downarrow_2 v_2 \text{ and } v_1 \sim v_2.\end{array} \tag{4.7}$$

We may stick to the definition (4.5) when we want to be sure that a compiler produces a correct output, but we only believe that some output will be produced in the first place.

4.1.3 Certifying compilers

Another approach to the secure compilation are so called *certifying* compilers. If we may find a property Δ which states that the compiler output is correct w.r.t. the compiler input, it is enough to verify whether the output satisfies Δ, and we just need to certify the verifier.

More formally, a certifying compiler is a pair of functions: $f : L_1 \hookrightarrow L_2$ and $g : L_1 \times L_2 \to \{\mathbf{true}, \mathbf{false}\}$, where f may be partial, such that the following has been proven (preferably in a proof system like Coq):

$$\begin{array}{l}\text{For all } e_1, e_2, \text{ if } g(e_1, e_2) = \mathbf{true} \text{ then for all } v_1, \text{ if } \mathit{init}_1(e_1) \Downarrow_1 v_1 \\ \text{then there exists } v_2 \text{ such that } \mathit{init}_2(e_2) \Downarrow_2 v_2 \text{ and } v_1 \sim v_2.\end{array} \tag{4.8}$$

Any certifying compiler may be easily transformed into a certified compiler in a sense of the definition (4.5). For a certifying compiler (f, g), we define a certified compiler f'

as follows:

$$f'(e) = \begin{cases} f(e) & \text{if } f(e) \text{ is defined and } g(e, f(e)) = \textbf{true}, \\ \mathit{undefined} & \text{otherwise.} \end{cases}$$

From (4.8) it easily follows that f' is certified.

4.2 Case study: reverse Polish notation

Now we go back to the example of compiling arithmetic expressions to reverse Polish notation. For simplicity, we consider only addition and multiplication of natural numbers.

Let $\Sigma = \{\oplus, \odot\}$ be a signature containing binary symbols and $\mathcal{T}$ be a set of terms built using natural numbers and symbols from Σ. Interpreting the symbols as addition and multiplication, we may define a semantics $\Downarrow_{\mathrm{A}} \subseteq \mathcal{T} \times \mathbb{N}$ with the following inference rules:

$$n \Downarrow_{\mathrm{A}} n, \text{ where } n \in \mathbb{N} \qquad \frac{t_1 \Downarrow_{\mathrm{A}} n \quad t_2 \Downarrow_{\mathrm{A}} m}{t_1 \oplus t_2 \Downarrow_{\mathrm{A}} n + m} \qquad \frac{t_1 \Downarrow_{\mathrm{A}} n \quad t_2 \Downarrow_{\mathrm{A}} m}{t_1 \odot t_2 \Downarrow_{\mathrm{A}} nm}$$

On the other hand, let $RPN = (\Sigma \cup \mathbb{N})^*$ be the set of all reverse Polish notation expressions. We may define semanitcs for RPN by a virtual machine, that is a relation $\rightarrow_{\mathrm{RPN}} \subseteq (RPN \times \mathbb{N}^*) \times (RPN \times \mathbb{N}^*)$:

$$(n{::}w, s) \rightarrow_{\mathrm{RPN}} (w, n{::}s), \text{ where } n \in \mathbb{N} \qquad (\oplus{::}w, n{::}m{::}s) \rightarrow_{\mathrm{RPN}} (w, (n+m){::}s)$$

$$(\odot{::}w, n{::}m{::}s) \rightarrow_{\mathrm{RPN}} (w, (nm){::}s) \qquad \frac{c_1 \rightarrow_{\mathrm{RPN}} c_2 \quad c_2 \rightarrow_{\mathrm{RPN}} c_3}{c_1 \rightarrow_{\mathrm{RPN}} c_3}$$

The initial configuration for the $\Downarrow_{\mathrm{A}}$ semantics is an identity function, while for the $\rightarrow_{\mathrm{RPN}}$ semantics it is defined as: $\mathit{init}_{RPN}(w) = (w, \varepsilon)$. The $\sim$ relation of equivalence of final values is a function: $\sim(n) = (\varepsilon, [n])$. The compilation function may be defined by the following equations:

$$\begin{aligned} f(n) &= [n] \\ f(t_1 \oplus t_2) &= f(t_1) +\!\!+ f(t_2) +\!\!+ [\oplus] \\ f(t_1 \odot t_2) &= f(t_1) +\!\!+ f(t_2) +\!\!+ [\odot] \end{aligned} \tag{4.9}$$

To prove that the property (4.2) holds for f, we use the following invariant:

Theorem 12 (Invariant)**.** *For all $e \in \mathcal{T}$ and $n \in \mathbb{N}$, if $e \Downarrow_A n$ then for all $g \in RPN$ and $s \in \mathbb{N}^*$, $(f(e) +\!\!+ g, s) \rightarrow_{RPN} (g, n{::}s)$.*

Proof. Induction on the structure of e. Trivial, if $e \in \mathbb{N}$. If $e = t_1 \oplus t_2$ then the invariant to prove is: $(f(t_1) +\!+ f(t_2) +\!+ [\oplus] +\!+ g, s) \rightarrow_{\text{RPN}} (g, n::s)$. Since $e \Downarrow_{\text{A}} n$, then there exist m_1, m_2 such that $t_1 \Downarrow_{\text{A}} m_1$ and $t_2 \Downarrow_{\text{A}} m_2$ and $m_1 + m_2 = n$. The induction hypothesis for t_1 gives us: $(f(t_1) +\!+ f(t_2) +\!+ [\oplus] +\!+ g, s) \rightarrow_{\text{RPN}} (f(t_2) +\!+ [\oplus] +\!+ g, m_1::s)$, and for t_2: $(f(t_2) +\!+ [\oplus] +\!+ g, m_1::s) \rightarrow_{\text{RPN}} ([\oplus] +\!+ g, m_2::m_1::s)$. Obviously $([\oplus] +\!+ g, m_2::m_1::s) \rightarrow_{\text{RPN}} (g, n::s)$, and, since $\rightarrow_{\text{RPN}}$ is transitive, we obtain the case. The case $e = t_1 \odot t_2$ is similar. □

If we take ε for g and s in the theorem, we get the desired property.

4.2.1 Formalization in Coq

First we define arithmetic expressions with three constructors: unary `Const` for natural numbers and one binary constructor for each operator (`Add` for $\oplus$ and `Mult` for $\odot$). Then, we define the natural semantics, one constructor of `sem` datatype for each rule. Both definitions are as follows:

```
Module A.

Inductive expr : Set :=
| Const : nat → expr
| Add :  expr → expr → expr
| Mult : expr → expr → expr.

Inductive sem : expr → nat → Prop :=
| S_Const : forall (n : nat),
    sem (Const n) n
| S_Add : forall (e1 e2 : expr) (n1 n2 : nat),
    sem e1 n1 → sem e2 n2 → sem (Add e1 e2) (n1 + n2)
| S_Mult : forall (e1 e2 : expr) (n1 n2 : nat),
    sem e1 n1 → sem e2 n2 → sem (Mult e1 e2) (n1 * n2).

End A.
```

The representation of RPN expressions is also obvious. We define a datatype for the $\oplus$ and $\odot$ symbols and numerical constants. Expressions are lists of such elements. A stack is a list of natural numbers and a configuration is a pair containing an expression and a stack. The rules for the abstract machine are represented by a `sem` datatype. The whole module is defined as follows:

```
Module RPN.

Inductive elem : Set :=
| Const : nat → elem
| Add : elem
| Mult : elem.

Definition expr := list elem.
Definition stack := list nat.
Definition config := (expr * stack)%type.

Inductive sem : config → config → Prop :=
| S_Push : forall (n : nat) (e : expr) (ns : stack),
    sem (Const n :: e, ns) (e, n :: ns)
| S_Add : forall (n m : nat) (e : expr) (ns : stack),
    sem (Add :: e, n :: m :: ns) (e, (n + m) :: ns)
| S_Mult : forall (n m : nat) (e : expr) (ns : stack),
    sem (Mult :: e, n :: m :: ns) (e, (n * m) :: ns)
| S_Trans : forall (c1 c2 c3 : config),
    sem c1 c2 → sem c2 c3 → sem c1 c3.

End RPN.
```

The compiler is defined by equations identical to (4.9). The proof of Theorem 12 is similar to the paper version:

```
Module To_RPN.

Fixpoint compile (e : A.expr) : RPN.expr :=
match e with
| A.Const n ⇒ RPN.Const n :: nil
| A.Add e1 e2 ⇒ compile e1 ++ compile e2 ++ RPN.Add :: nil
| A.Mult e1 e2 ⇒ compile e1 ++ compile e2 ++ RPN.Mult :: nil
end.

Theorem invariant :
forall (e : A.expr) (n : nat), A.sem e n →
  forall (f : RPN.expr) (ns : RPN.stack),
  RPN.sem (compile e ++ f, ns) (f, n :: ns).
Proof with intros; simpl; eauto.
intros ? ? H.
```

```
induction H... (* solves Const *)
(* Add *)
repeat rewrite app_ass.
eapply RPN.S_Trans...
eapply RPN.S_Trans...
rewrite plus_comm...
(* Mult *)
repeat rewrite app_ass.
eapply RPN.S_Trans...
eapply RPN.S_Trans...
rewrite mult_comm...
Qed.

Theorem completeness :
forall (e : A.expr) (n : nat),
  A.sem e n →
  RPN.sem (compile e, nil) (nil, n :: nil).
Proof with auto.
intros.
cutrewrite (compile e = compile e ++ nil).
apply invariant...
apply app_nil_end.
Qed.

End To_RPN.
```

The whole Coq development is available in the file `rpn/rpn.v`.

5 Implementation of the STG machine

The STG machine is a semantics for the STG language – and so Haskell itself – which defines a step-by-step evaluation of STG expressions. We may say that it is imperative, because each step modifies the state of the machine, which is a tuple, and thus elements that are modified are the heap, the current expression, the environment or the stacks.

The STG machine is an *abstract* machine (opposed to *virtual* machines). It means that an expression of the STG language is a part of the state, and in each step of evaluation it can be replaced by an other expression, may it be a subexpression of the current expression, an expression from the heap or an alternative from the stack. On the other hand, virtual machines contain a constant code store generated (compiled) from the source expression, which cannot be modified during evaluation. The idea of a virtual machine equivalent to the STG machine may be essentially useful in designing a compiler for Haskell, because it is sufficient to generate a piece of code in the low-level target language for each element of the code store. Each piece modifies other elements of the state according to the rules for transitions of the machine. Additionally, expressions that are elements of closures or are held in the stack of continuations may be replaced with equivalent pointers to the code store.

Another issue is management of environments. It seems inefficient to hold a map from variables to addresses at the runtime. As will be seen, by a trivial compile-time analysis and different behavior of the stacks, we may replace the environments on the heap by plain tuples of addresses, and simplify the environment of the current expression to a stack of addresses (which additionally can be merged with the argument stack).

Another thing that may be difficult to implement in a compiler is the argument stack being held as an element of the stack of continuations. We will see that, by a simple trick, we can make size of elements on the continuation stack constant.

This chapter proposes a series of extensions of the STG language along with appropriate semantics. The series leads to a machine that has a model of computation very similar to the STG machine: we start with a semantics very close to the explicit-environment semantics, make elements of its configurations more compile-friendly and then use the D-CPS transformation to obtain an abststract machine. Then, a procedure to 'flatten' expressions into code store is proposed, thus making the abstract machine virtual. More precisely:

1. The first language is called $\mathcal{C}$ (for *closures*). It enables tuples of arguments to be elements of closures, instead of environments. Thanks to the use of de Bruijn

indices, we formalize environments as stacks, which solves part of the previously addressed issues.

2. The second language is called $\mathcal{CU}$ (for *clean up*). It introduces one global environment instead of environments bound to current expressions. Such current environments had to be contained on the stack to be restored when returning to a continuation, while in $\mathcal{CU}$ we return to a suitable environment by a simple 'cleaning' procedure (hence the name).

3. The third language is called $\mathcal{FB}$. It stands for *fake bottom*, which is a pointer to an element on the argument stack, which is seen as the last one when performing the argument check. Now it is sufficient to store the fake bottom in the continuation, not the whole argument stack.

4. The next language is called $\mathcal{CCP}$ (for *current closure pointer*). When entering a closure, we do not need to affix it (in practice: copy) to the environment. Instead we access it through its address on the heap.

5. The fifth language is called $\mathcal{D}$ (for *D-CPS*). We perform the transformation and obtain an abstract machine.

6. The last language is called $\mathcal{VM}$ (for *virtual machine*). To get a code store, we gather all the subexpressions of the source program and create supplementary instructions that are equivalent to all expressions that may be created during updates.

Each language, semantics and compilation procedure is formalized in Coq. Appropriate proofs of equivalence are given, resulting in a certified compiler from $\mathcal{C}$ to $\mathcal{D}$, and a certifying compiler from $\mathcal{D}$ to $\mathcal{VM}$.

We see a modular compiler as a composition of simple compilers between input, intermediate and output languages. In case of certified compilers these modules provide also proofs of correctness of compilation procedures. If two languages has the same syntax but different semantics, the compilation procedure may be an identity function, but, for the proof of equivalence of the semantics, the whole module is not negligible. It is the case here, since the languages $\mathcal{CU}$ to $\mathcal{D}$ all use the same syntax, and compilation procedures in-between are all identities.

It is important that the languages $\mathcal{CU}$, $\mathcal{FB}$ and $\mathcal{CCP}$ all have natural semantics. Those languages exist to make states of the corresponding (via the D-CPS transformation) virtual machines lighter by using pointers and a global environment. The correctness of compilation is easier to prove in Coq for natural semantics. That is why they are studied before the D-CPS transformation.

5.1 Commons and Closures

We start with a language called $\mathcal{C}$. The name is a short for *closures*, because it differs from STG by management of closures. It allows us to simplify the concept of environment.

5.1.1 Free variables, closures

The $\mathcal{C}$ language uses a very simple idea present in the original STG paper by Peyton-Jones: for each lambda-form we enlist its free variables. For example, the STG lambda form $(\lambda_n\, x\, y.\, C_0\, x\, y\, z\, t)$ becomes $(z\, t\, \lambda_n\, x\, y.\, C_0\, x\, y\, z\, t)$ in the $\mathcal{C}$ language.[1].

In the STG semantics, the free variables always refer (by the environment) to addresses stored in the closure. The same holds here, because the order of those variables is the same as the order of the corresponding addresses when the lambda form is allocated on the heap, so there is no need for an indirection created by the environment. If the variable t is second on the list before λ, it means that it refers to the second slot in the closure, which now can be stored as a plain tuple, not an environment. For an example, see Figure 5.1.

An STG closure		
$\lambda_n\, x\, y.\, C_0\, x\, y\, z\, t$	$z \mapsto a_0$	$t \mapsto a_1$

A $\mathcal{C}$ closure		
$z\, t\, \lambda_n\, x\, y.\, C_0\, x\, y\, z\, t$	a_0	a_1

Figure 5.1: Representation of closures on the heap

This representation of free variables becomes useful when we use de Bruijn indices. It is crucially important that this time we name variables in a multiple binder **from left to right**. So an alternative $C_0\, x_1\, x_2\, x_3 \to C_1\, x_1\, x_2\, x_3$ becomes $C_0[3] \to C_1\, 0\, 1\, 2$. Of course outer binders still bind greater indices than inner binders.

We use two kinds of indices: first that point to some bound variables (bound by a lambda-form, by a definition in a letrec expression or by an alternative) and other that point to a slot in the closure. We call the former *bound* indices (B for short), while the latter are dubbed *closure* indices (C for short). Note that the list of variables before λ is now used only in the moment when the lambda-form is allocated – when addresses are gathered respectively to the list and the tuple to be allocated on the heap is created – and differ from the variables inside the lambda-form, which are now not free (in the sense of 'sticking out' over the number of surrounding binders), but are of another type. Figures 5.2 and 5.3 show an example.[2]

[1] Do not confuse this with lambda-lifting [22] The variables before λ are not bound there, they just give the free variables of a lambda-form an order in which they are placed in a closure on the heap.

[2] Remember that the indices in the environment refer to the whole lambda-form, thus an index 1 is represented inside the lambda-forms as 3, because the lambda-forms bind two variables. Also, since the list before λ is used only on allocation and variables from that list are not referred to inside the lambda-form, we skip the list in the figure 5.3.

$$\emptyset : \left(\begin{array}{ll} \textbf{let} & \lambda_u[0].\, e_0 \\ & \lambda_n[5].\, e_1 \\ & \lambda_n[2].\, e_2 \\ \textbf{in} & \textbf{let } \lambda_n[2].\, C_0\ 0\,1\,3\,4 \textbf{ in } e_3 \end{array} \right) \quad \Rightarrow$$

$$\left(\begin{array}{ll} a_0 \mapsto & \boxed{\lambda_u[0].\, e_0 \mid \ldots} \\ a_1 \mapsto & \boxed{\lambda_n[5].\, e_1 \mid \ldots} \\ a_2 \mapsto & \boxed{\lambda_n[2].\, e_2 \mid \ldots} \end{array} \right) : \textbf{let } \lambda_n[2].\, C_0\ 0\,1\,3\,4 \textbf{ in } e_3 \quad \Rightarrow$$

$$\left(\begin{array}{ll} a_0 \mapsto & \boxed{\lambda_u[0].\, e_0 \mid \ldots} \\ a_1 \mapsto & \boxed{\lambda_n[5].\, e_1 \mid \ldots} \\ a_2 \mapsto & \boxed{\lambda_n[2].\, e_2 \mid \ldots} \\ a_3 \mapsto & \boxed{\lambda_n[2].\, C_0\ 0\,1\,3\,4 \mid 1 \mapsto a_1 \mid 2 \mapsto a_2} \end{array} \right) : e_3$$

Figure 5.2: Representation of closures on the heap using de Bruijn indices in STG

$$\emptyset : \left(\begin{array}{ll} \textbf{let} & \lambda_u[0].\, e_0 \\ & \lambda_n[5].\, e_1 \\ & \lambda_n[2].\, e_2 \\ \textbf{in} & \textbf{let } B1\, B2\, \lambda_n[2].\, C_0\ B0\, B1\, C0\, C1 \textbf{ in } e_3 \end{array} \right) \quad \Rightarrow$$

$$\left(\begin{array}{ll} a_0 \mapsto & \boxed{\lambda_u[0].\, e_0 \mid \ldots} \\ a_1 \mapsto & \boxed{\lambda_n[5].\, e_1 \mid \ldots} \\ a_2 \mapsto & \boxed{\lambda_n[2].\, e_2 \mid \ldots} \end{array} \right) : \textbf{let } B1\, B2\, \lambda_n[2].\, C_0\ B0\, B1\, C0\, C1 \textbf{ in } e_3 \quad \Rightarrow$$

$$\left(\begin{array}{ll} a_0 \mapsto & \boxed{\lambda_u[0].\, e_0 \mid \ldots} \\ a_1 \mapsto & \boxed{\lambda_n[5].\, e_1 \mid \ldots} \\ a_2 \mapsto & \boxed{\lambda_n[2].\, e_2 \mid \ldots} \\ a_3 \mapsto & \boxed{\lambda_n[2].\, C_0\ B0\, B1\, C0\, C1 \mid a_1 \mid a_2} \end{array} \right) : e_3$$

Figure 5.3: Representation of closures on the heap using de Bruijn indices in $\mathcal{C}$

The list before λ may consist of both kinds of variables. They are B if bound by the immediate outer lambda-form, an alternative or 'name' of the definition[3] in a letrec expression in the immediate outer lambda-form. They are C if they are free in the immediate outer lambda-form.

The definition of such two-kind variables in Coq is done as follows:

```
Inductive var : Set :=
| B_ind : nat → var
| C_ind : nat → var.
Definition vars := list var.
```

We also define generic lambda-forms, which are parametrized with a language of the body. It is useful to use one definition of lambda-forms in all languages, because the shape of lambda-forms does not change, so we may count on some generic theorems:

```
Definition bind := nat.
Inductive upd_flag : Set := Update | Dont_update.
Inductive gen_lambda_form {Expr : Set} : Set :=
| Lf : upd_flag → vars → bind → Expr → gen_lambda_form.
```

5.1.2 Environments

If a letrec expression defines n lambda-forms, then all the variables 'after' the expression (inside the 'in' part) are represented by indices shifted by n from 'before' the whole letrec expression. Not incidentally, in the STG semantics during allocation we add n elements to the environment. The same applies to alternatives (when we choose an alternative, each variable adds a new element to the environment), and when entering a closure, we add to the environment (at first consisting only of the environment from the closure) one element for each bound variable. See Figure 5.4 for an example with variables and indices (assume that x evaluates to $C_0\,x\,y$ in the environment $[x \mapsto a_0, y \mapsto a_1]$).

Passing through n binders shifts the environment by n and adds the following to the environment:

$0 \mapsto a_0$	$\cdots$	$(n-1) \mapsto a_n$

If we start with an empty environment, we will always get an environment

$0 \mapsto \ldots$	$1 \mapsto \ldots$	$\cdots$	$n \mapsto \ldots$

[3] As in the following:

$$\begin{array}{ll} \textbf{let} & x = \ldots \\ & y = x\,\lambda_\pi\ \ldots \\ \textbf{in} & \ldots \end{array}$$

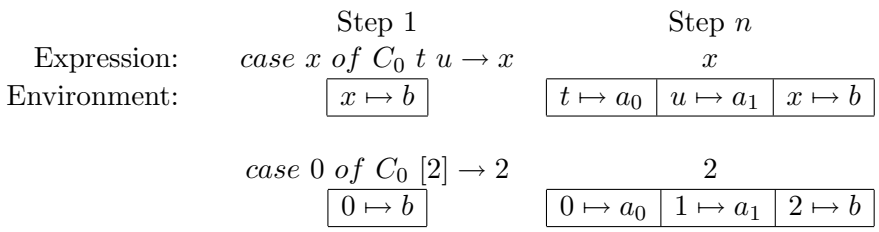


Figure 5.4: Passing through binders and environment allocation in STG (normal and de Bruijn notation)

An index k will always refer to the k-th element of the environment. This is why in $\mathcal{C}$ we do not think of the environment as of a map, but as of a stack (see Figure 5.5), where an index n refers to n-th element of the stack.

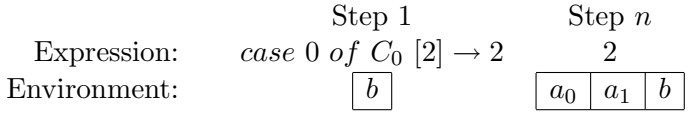


Figure 5.5: Passing through binders and environment allocation in $\mathcal{C}$

It all works fine if we have only B indices, but what about C indices? The solution is: we always hold two stacks, B and C. Entering a closure puts its tuple of addresses as the current C-stack (which will not be modified until the next entering). Actual arguments for the lambda-form are put on the B-stack, which may be later extended when reaching a letrec expression or choosing an alternative. Since each variable is a B or C index, it is obvious to which stack it refers.

We also distinguish another entity: addresses on the heap. In STG they are just variables, which is an inheritance from the semantics with substitutions. There is no reason for such confusion now.

The Coq definition of environment is trivial. For simplicity, we use the type `nat` for addresses, though from the point of view of semantics it may be any infinite set with decidable equality.

```
Definition address : Set := nat.
Definition addresses : Set := list address.
Definition env : Set := addresses.
```

We also define two auxiliary functions: `look_up` looks up a value of a variable in a two-environment setting (the argument `e` is a B-stack and `a` is a C-stacks), and `map_env` is a monadic mapping of `look_up` on a list of variables:

```
Definition look_up (e : env) (a : addresses) (v : var) :
  option address :=
match v with
| B_ind n ⇒ nth n e
| C_ind n ⇒ nth n a
end.

Fixpoint map_env (e : env) (a : addresses) (vs : vars)
  {struct vs} : option addresses :=
match vs with
| nil ⇒ Some nil
| x::xs ⇒
  match map_env e a xs with
  | Some ps ⇒
    match look_up e a x with
    | Some p ⇒ Some (p::ps)
    | None ⇒ None
    end
  | None ⇒ None
  end
end.
```

5.1.3 Commons

The above is common for most of the considered languages. In this section we present other common constructions.

Constructors and alternatives

Each constructor name is a `nat`. It is important since the right alternative in the semantics of case expressions is chosen using the `nth : forall (A : Set), nat -> list A -> option A` function, so alternatives do not contain a constructor name any longer, it is an order of alternatives in a case expression that matters. Thus all 'datatypes' (quotes because the STG and $\mathcal{C}$ languages do not provide explicit datatype declarations) must consist of constructors being a prefix of a standard order on natural numbers, so a list may consist of two constructors: a constant 0 and a binary 1. Obviously, constructor names are shared between datatypes, 1 sometimes means `Cons` and sometimes `True`.

We also introduce a generic alernative, similar to the generic lambda-form:

```
Definition constructor := nat.
Inductive gen_alt {Expr : Set} : Set :=
| Alt : bind → Expr → gen_alt (Expr := Expr).
```

Heaps

For different semantics we also need a generic heap with some operations with an obvious semantics:

```
Definition gen_heap {Clo : Set} : Set := address → option Clo.

Definition set {Clo : Set} (g : gen_heap) (a : address)
  (clos : Clo) : gen_heap := (...)

Definition alloc {Clo : Set} (g : gen_heap) (addrs : addresses)
  (clos : list Clo) : gen_heap := (...)
```

Black holes again

The semantics for implementations use black holes, which are used mostly in proofs and can be eliminated in later languages. A datatype `node` represents an element which is the first element in pairs constructing closures. It can be either a `Node` containing a lambda-form or a black hole. Note that a black-holed (aka *clogged*) closure on a heap still contains its second element, that is a tuple of addresses.

```
Inductive node {Expr : Set} : Set :=
| Node : gen_lambda_form (Expr := Expr) → node
| Black_hole : node.

Definition gen_closure {Expr : Set} : Set :=
(node (Expr := Expr) * addresses)%type.

Definition clog {Expr : Set}
  (g : gen_heap (Clo := gen_closure (Expr := Expr)))
  (a : address) : gen_heap (Clo := gen_closure) :=
fun (x : address) ⇒ if eq_nat_dec x a
  then match g x with
    | None ⇒ None
    | Some (_, addrs) ⇒ Some (Black_hole, addrs) end
  else g x.
```

For sake of simplicity, we identify lambda-forms with `Node` closures:

```
Definition coer_lf_closure_type {Expr : Set}
  (lf : gen_lambda_form (Expr := Expr)) : node := Node lf.
Coercion coer_lf_closure_type : gen_lambda_form >-> node.

Definition Lf_n {Expr : Set} : vars → bind → Expr → node :=
Lf (Expr := Expr) Dont_update.

Definition Lf_u {Expr : Set} : vars → bind → Expr → node :=
Lf (Expr := Expr) Update.

Definition gen_closures {Expr : Set} : Set :=
list (gen_closure (Expr := Expr)).
```

5.1.4 The language $\mathcal{C}$ and its natural semantics

The language $\mathcal{C}$ is different from STG only by the explicit free variables in lambda-forms, but in Coq it is defined using generic lambda-forms and alternatives:

```
Inductive expr : Set :=
| App : var → vars → expr
| Constr : constructor → vars → expr
| Letrec : list (gen_lambda_form (Expr := expr)) → expr → expr
| Case : expr → list (gen_alt (Expr := expr)) → expr.
```

The semantics

The semantics relies heavily on the formalization (mostly on de Bruijn indices), thus we present it only in Coq.

First, we define types for closures that will be held on the heap, and the type of the heap itself:

```
Definition closure := gen_closure (Expr := expr).
Definition heap := gen_heap (Clo := closure).
```

Another novelty is that we use two kinds of configurations: `Config` and `Enter`. They work similar to the **eval** and **enter** instruction of the STG machine, the only difference is that we introduce them in the last step of derivation of the STG machine, while here we do it at the beginning. The `Enter` configurations are used for applications after accumulating arguments in the argument stack. We do it to put the semantics in order,

since entering a closure does not really need an expression or an environment, but an address of a to-be-entered closure.

Both kinds of configurations contain a heap and an argument stack. The `Config` configuration contains additionally an expression and its environment, which is represented by an `env` item for B indices and a tuple (represented by a list of addresses, since for each closure size of the tuple may differ) for C indices. The `Enter` configuration contains (except for the heap and the argument stack) only an address of the closure to be entered.

```
Definition argstack := addresses.

Inductive config : Set :=
| Config : heap → expr → env → argstack → addresses → config
| Enter : heap → address → argstack → config.

Notation " << a , b , c , d , e >> " := (Config a b c d e)
                                        (at level 70).
Notation " << a , b , c >> " := (Enter a b c) (at level 70).
```

There are also two kinds of values: One for constructors and one for partial applications. The former contains a heap, a constructor name and a list of addresses, which model the constructor's arguments. The latter also contains a heap, an address of the closure (which could not be entered because of the lack of the number of arguments) and a list of addresses, which are the first n actual arguments of the partial application.

```
Inductive value : Set :=
| Val_con : heap → constructor → addresses → value
| Val_pap : heap → address → addresses → value.
```

The semantics is and inductive Coq datatype. Its type is: `config -> value -> Prop`:

```
Reserved Notation " a ↓ b " (at level 70, no associativity).

Inductive Sem : config → value → Prop :=
(...)
where " a ↓ b " := (Sem a b).
```

Now, we present rules of the semantics. We separately discuss each one.

Config rules

There are four rules for the `Config` kind of configurations, one for each constructor of the language.

The constructor expression evaluates to a value, but only if all arguments are in the environment and there are no pending arguments on the argument stack:

```
Inductive Sem : config → value → Prop :=
| Cons : forall Gamma C xs sigma clo addrs
  (MAPENV : map_env sigma clo xs = Some addrs),
  << Gamma, Constr C xs, sigma, nil, clo >> ↓
       Val_con Gamma C addrs
(...)
```

The rule for applications finds values for arguments in the environment and put them on the argument stack. Then it finds an address in the environment for the head of the application and enters:

```
| Accum : forall Gamma x xs sigma args clo p new_args Value
  (MAPENV  : map_env sigma clo xs = Some new_args)
  (LOOKUP  : look_up sigma clo x = Some p)
  (PREMISE : << Gamma, p, new_args++args >> ↓ Value),
  << Gamma, App x xs, sigma, args, clo >> ↓ Value
```

The rule for letrec expressions is more complicated. A separate procedure, named `make_closures`, creates the closures that need to be allocated on the heap. It is the place where variables before λ are replaced with corresponding (via environment) addresses. The procedure is called with a tuple of fresh addresses affixed to the current environment because the lambda-forms are mutually recursive and 'see' each other.

```
Definition make_closure {Expr : Set} (e : env) (a : addresses)
  (lf : gen_lambda_form (Expr := Expr)) : option gen_closure :=
match lf with
| Lf _ vs _ _ ⇒
  match map_env e a vs with
  | Some addrs ⇒ Some (Node lf, addrs)
  | None ⇒ None
  end
end.

Fixpoint make_closures {Expr : Set} (e : env) (a : addresses)
  (lfs : list (gen_lambda_form (Expr := Expr))) {struct lfs}
  : option gen_closures:=
match lfs with
| nil ⇒ Some nil
| l::lfs0 ⇒
  match make_closures e a lfs0 with
```

```
  | Some clos ⇒
    match make_closure e a l with
    | Some clo ⇒ Some (clo::clos)
    | None ⇒ None
    end
  | None ⇒ None
  end
end.

Inductive Sem : config → value → Prop :=
(...)
| Let : forall Gamma e lfs sigma current_clo closures Value
                addrs args
  (LENGTH  : length addrs = length lfs)
  (FRESH   : forall p : address, In p addrs → Gamma p = None)
  (CLOS    : make_closures (addrs++sigma) current_clo lfs =
               Some closures)
  (PREMISE : << alloc Gamma addrs closures, e, addrs++sigma,
                  args, current_clo >> ↓ Value),
  << Gamma, Letrec lfs e, sigma, args, current_clo >> ↓ Value
(...)
```

The rule for case expression is simple. It computes a value of the inner expression. If the value is `Val_con`, the correct alternative is chosen (n-th for a constructor name n). The addresses from `Val_con` are attached to the environment, and we evaluate the body of the alternative:

```
| Case_of : forall Gamma e als sigma clo Value bind Delta e0 C
                   addrs args
  (SELECT : nth C als = Some (Alt bind e0))
  (LENGTH : bind = length addrs)
  (LEFT   : << Gamma, e, sigma, nil, clo >> ↓
                Val_con Delta C addrs)
  (RIGHT  : << Delta, e0, addrs++sigma, args, clo >> ↓ Value),
  << Gamma, Case e als, sigma, args, clo >> ↓ Value
```

Enter rules

There are four rules for entering a closure.

If the argument check fails, that is there are less arguments on the argument stack than the closure would like to consume, the address of that closure and the content of

argument stack is the value:

```
| App1 : forall Gamma p args clovars bind e some_clo
  (ONHEAP : Gamma p = Some (Lf_n clovars bind e, some_clo))
  (LENGTH : bind > length args),
  << Gamma, p, args >> ↓ Val_pap Gamma p args
```

If there are enough arguments, we take the prefix of the argument stack of the desired length (number of arguments the closure would like to consume) as the current environment. Then, we evaluate the body of the closure:

```
| App2 : forall Gamma p args Value e bind clo clovars
  (ONHEAP  : Gamma p = Some (Lf_n clovars bind e, clo))
  (LENGTH  : bind <= length args)
  (PREMISE : << Gamma, e, firstn bind args, skipn bind args,
                clo >> ↓ Value),
  << Gamma, p, args >> ↓ Value
```

Note that by coercion `Lf_n...` really means `Node (Lf_n...)`. If there are black holes at the address `a` the rules do not apply. Also note that we do not clog closures that are non-updatable, because if they have some arguments, they may be safely reentered from the inside with other actual arguments, not leading to an infinite loop.

There are two rules for updatable closures. The first one states that if a closure evaluates to a constructor, the constructor is the value, but we also update the heap. We have to create a closure form the value, which is a very simple procedure shown in Figure 5.6.

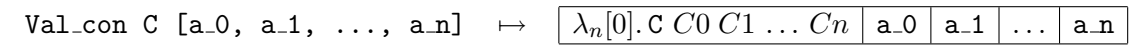


Figure 5.6: Creating a closure form a `Val_con` value

To implement it, we use a function `from a b` which creates a list of `b` successive natural numbers beginning with `a`:

```
Fixpoint from (a b : nat) {struct b} : list nat :=
match b with
| 0 ⇒ nil
| S m ⇒ a :: from (S a) m
end.

Definition make_cons (C : constructor) (addrs : addresses) :
  closure :=
(Lf_n nil 0 (Constr C (map C_ind (from 0 (length addrs)))),
  addrs).
```

In the moment of entering the closure, we clog the closure with a black hole, thus preventing it from being entered while evaluating the closure:

```
| App3 : forall Gamma p C addrs clovars Delta e clo
  (ONHEAP   : Gamma p = Some (Lf_u clovars 0 e, clo))
  (PREMISE : << (clog Gamma p), e, nil, nil, clo >> ↓
                  Val_con Delta C addrs),
  << Gamma, p, nil >> ↓
       Val_con (set Delta p (make_cons C addrs)) C addrs
```

If a closure evaluates to a constructor, we use the alternative version of update, that is we update the heap with the normal form created from the `Val_pap` value (see also Figure 5.7):

`Val_pap b [a_0, a_1, ..., a_n]` $\mapsto$

$\lambda_n[0].C0\,C1\,C2\,\ldots\,C(n+1)$	`b`	`a_0`	`a_1`	...	`a_n`

Figure 5.7: Creating a closure form a `Val_pap` value

```
Definition make_pap (q : address) (addrs : addresses)
  : closure :=
(Lf_n nil 0 (App (C_ind 0) (map C_ind (from 1 (length addrs)))),
  q::addrs).

Inductive Sem : config → value → Prop :=
(...)
| App4 : forall Gamma p args Value q addrs e clovars clo_e
                Delta
  (ONHEAP1 : Gamma p = Some (Lf_u clovars 0 e, clo_e))
  (LEFT     : << (clog Gamma p), e, nil, nil, clo_e >> ↓
                  Val_pap Delta q addrs)
  (RIGHT    : << set Delta p (make_pap q addrs), q,
                  addrs++args >> ↓ Value),
  << Gamma, p, args >> ↓ Value
(...)
```

5.2 Clean up

To justify the existence of the $\mathcal{CU}$ language, we analyze the `Case_of` constructor of the semantics for the $\mathcal{C}$ language. First, in the `LEFT` premise, we evaluate the expression `e` with `sigma` as the current environment. During the evaluation of `e`, the environment `sigma` may be modified, or even disposed of, if another closure is entered. Then, in the `RIGHT` premise, we evaluate the body of an appropriate alternative with `sigma` extended with addresses for variables bound by the pattern of the alternative as the current environment.

In a virtual machine we would have to somehow save `sigma` while evaluating `e`, for example on a stack, to recall it when we start evaluating the body of the alternative. This operation would be costly and our goal is to obtain constant size of elements on all stacks of the machine.

Our solution is to keep one global environment instead of a series of local environments. Notice that in the semantics for $\mathcal{C}$, environments are built when evaluating an expression, and then cleared while entering another closure. The idea is to build one environment on top of another, and clean it up not when entering a new closure, but when leaving the old one (it may seem unexpected, but there is a difference). Since evaluation of an expression inside a case expression will clean up only its 'mess', the right environment will be restored before returning to the appropriate alternative.

5.2.1 Leaving points

Leaving points of an expression are those positions, on which $\mathcal{C}$ semantics clears its current environment, that is when a subexpression immediately evaluates to a value, enters another closure or selects a correct alternative. Obviously, they are applications and constructors.

When the evaluation reaches a leaving point, it is certain that the current environment will be cleared in the next step, because values do not store an environment, entering a closure replaces the environment with values from the argument stack, and selecting an appropriate alternative restores the environment from before the evaluation of the case expression.

In the $\mathcal{CU}$ language we introduce one global environment. Instead of disposing of the whole environments, each expression will clear only the part of the global environment which was built during its evaluation. This cleaning procedure must be done in a leaving point, since it is easy to compute how many elements were put on the environment during the evaluation of the current expression. For each leaving point this number is constant, but may vary for different leaving points of the same expression. We call such number the *weight* of the leaving point.

The computation of the weight of a leaving point is based on a practical principle: clean up after yourself. If we enter a closure, before we leave it, we clean all the items

that were put on the environment in between. If we evaluate an expression inside a case expression, before we choose the right alternative, we clean up everything that was put on the environment in between. Since the environment stores values for variables introduced by outer binders (see Section 5.1.2), the weight of an expression is the number of bound variables from a certain point of the expression. These points are:

- Top of the outermost expression for a leaving point of that expression,
- The binder of arguments of a lambda-form for a leaving point of that lambda-form,
- The case expression for an expression in between *case* and *of* keywords.

Figure 5.8 shows an example expression with weights in parenthesis. For an example of manipulation of the global environment, see Figure 5.9. Assume that f completes its evaluation on a laving point C with weight 2. The b_0 and b_1 are addresses stored in the environment during the evaluation of f, which are removed from the environment when reaching the leaving point C (which of course does not have to be a leaving point of f if another closure was entered meanwhile).

$$
\begin{array}{rl}
\textbf{letrec} & \mathrm{f} = \lambda_n\, x\, y.\ x\, y\ (2) \\
\textbf{in} & \textbf{case letrec } g = \lambda_n\, x\, y.\ x\, y\ (2) \textbf{ in } C_1\ (1) \textbf{ of} \\
& \quad C_0\, x\, y\, z \to f\, x\, z\ (4) \\
& \quad C_1 \to C_0\, f\, f\, f\ (1)
\end{array}
$$

Figure 5.8: Example of leaving points with weights in an expression

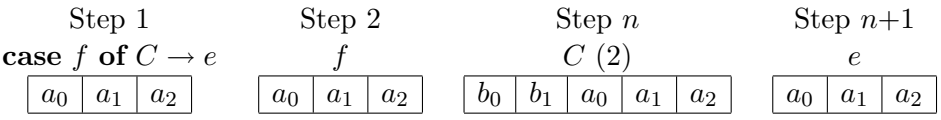


Figure 5.9: Cleaning up the global environment in an evaluation of a case expression

5.2.2 The language $\mathcal{CU}$ and its semantics

The definition of the $\mathcal{CU}$ language is very similar to $\mathcal{C}$, except for additional arguments in `App` and `Constr`, which represent the weights:

```
Definition cleanup := nat.
Inductive expr : Set :=
| App : var → vars → cleanup → expr
| Constr : constructor → vars → cleanup → expr
| Letrec : list (gen_lambda_form (Expr := expr)) → expr → expr
| Case : expr → list (gen_alt (Expr := expr)) → expr.
```

Since the environment is now global, it has to be a part of values and `Enter` configurations now, just like the heap and the argument stack:

```
Inductive config : Set :=
| Config : heap → expr → env → argstack → addresses → config
| Enter : heap → address → argstack → env → config.

Notation " << a , b , c , d , e >> " := (Config a b c d e)
                                        (at level 70).
Notation " << a , b , c , d >> " := (Enter a b c d)
                                    (at level 70).
```

For the sake of simplicity (though this may be arguable) we decided to leave the `value` datatype as it were, and add the global environment to the type of semantics:

```
Inductive Sem : config → (value * env) → Prop := (...)
```

We do not show the whole semantics here (refer to the source code for details), but only four constructors illustrating the issue:

```
Reserved Notation " a ↓ b " (at level 70, no associativity).

Inductive Sem : config → (value * env) → Prop :=

| Cons : forall Gamma C xs sigma clo addrs cleanup
  (MAPENV : map_env sigma clo xs = Some addrs),
  << Gamma, Constr C xs cleanup, sigma, nil, clo >> ↓
       (Val_con Gamma C addrs, (skipn cleanup sigma))

| Accum : forall Gamma x xs sigma args clo p new_args Value
                 cleanup
  (MAPENV  : map_env sigma clo xs = Some new_args)
  (LOOKUP  : look_up sigma clo x = Some p)
  (PREMISE : << Gamma, p, new_args++args,
                  skipn cleanup sigma >> ↓ Value),
  << Gamma, App x xs cleanup, sigma, args, clo >> ↓ Value

| App2 : forall Gamma p args Value e bind clo clovars sigma
  (ONHEAP  : Gamma p = Some (Lf_n clovars bind e, clo))
  (LENGTH  : bind <= length args)
  (PREMISE : << Gamma, e, firstn bind args ++ sigma,
                  skipn bind args, clo >> ↓ Value),
  << Gamma, p, args, sigma >> ↓ Value
```

```
| Case_of : forall Gamma e als sigma clo Value bind Delta e0
                   C addrs args theta
  (SELECT : nth C als = Some (Alt bind e0))
  (LEFT   : << Gamma, e, sigma, nil, clo >> ↓
                (Val_con Delta C addrs, theta))
  (RIGHT  : << Delta, e0, addrs++theta, args, clo >> ↓ Value),
  << Gamma, Case e als, sigma, args, clo >> ↓ Value

(...)
where " a ↓ b " := (Sem a b).
```

Just like in the $\mathcal{C}$ semantics, the `Cons` rule returns the constructor from the expression as a value, but it additionally cleans up the global environment according to its weight. The same holds for `Accum`, which, before moving to an `Enter` configuration, does the cleaning.

`Accum` is used as the last rule in an old closure. In the premise there will be one of the `App` rules, which are responsible for entering a new closure. They cannot control from which expression they were entered, so they cannot do the cleaning, and they reasonably assume that the environment is ready for new allocations. It is shown in the `App2` rule, which put the actual arguments of the closure on the global environment.

The `Case_of` rule does not restore the environment from before the evaluation of `e`, because it, just as reasonably, assume that `e` cleaned up the environment after its evaluation.

5.2.3 Certified compiler $\mathcal{C} \rightarrow \mathcal{CU}$

To compile a $\mathcal{C}$ expression into a $\mathcal{CU}$ expression, we need to implement the previously stated rules (the three bullets in Section 5.2.1):

```
Fixpoint compute_cleanup_n (n : nat) (e : C.expr) {struct e} :
  CU.expr :=
match e with
| C.App x xs ⇒ CU.App x xs (n)
| C.Constr C xs ⇒ CU.Constr C xs (n)
| C.Letrec lfs f ⇒ CU.Letrec
    (map (fun lf ⇒ match lf with
       Lf pi xs bind e ⇒ Lf pi xs bind
                                (compute_cleanup_n bind e)
    end) lfs)
    (compute_cleanup_n (length lfs + n) f)
| C.Case f als ⇒ CU.Case (compute_cleanup_n 0 f)
    (map (fun alt ⇒ match alt with
```

```
      Alt bind e ⇒ Alt bind (compute_cleanup_n (bind + n) e)
    end) als)
end.

Definition compile : C.expr → CU.expr := compute_cleanup_n 0.
```

We define a function `promote_heap`, which maps the function `compile` on lambda-forms on the heap, and `promote_value`, which applies `promote_heap` to the heap in a value. A value and a 'promoted' value are equivalent in a sense, that the latter is the former extended only by weights of expressions. This allows us to formulate and prove the following robust invariant:

```
Definition clo_cleanup (clo : C.closure) :=
match clo with
| (Node (Lf pi xs bind e), vs) ⇒
    (Lf pi xs bind (compute_cleanup_n bind e) : node, vs)
| (Bloack_hole, addrs) ⇒ (Black_hole, addrs)
end.

Definition promote_heap (g : C.heap) : CU.heap :=
(heap_map clo_cleanup g).

Definition promote_value (v : C.value) : CU.value :=
match v with
| C.Val_con Gamma C ps ⇒ CU.Val_con (promote_heap Gamma) C ps
| C.Val_pap Gamma p ps ⇒ CU.Val_pap (promote_heap Gamma) p ps
end.

Lemma invariant :
forall Config Value
  (EVAL : C.Sem Config Value),
  match Config with
  | C.Config Gamma e sigma args clo ⇒
      forall (n : nat) (tau : env),
      CU.Sem (CU.Config (promote_heap Gamma)
             (compute_cleanup_n n e) (sigma++tau) args clo)
             (promote_value Value, skipn n (sigma++tau))
  | C.Enter Gamma p ps ⇒
      forall (tau : env),
      CU.Sem (CU.Enter (promote_heap Gamma) p ps tau)
             (promote_value Value, tau)
  end.
```

The invariant holds for all $\mathcal{C}$ configurations thus `compile` is a certified compiler to $\mathcal{CU}$. Note that the compiler is not sound. First of all, the $\mathcal{CU}$ expressions may use some elements of the environment which in the $\mathcal{C}$ semantics were lying on a forgotten (or stored on a stack in case of an abstract machine) environment and the element would not be found by `C.look_up` (which may be a case only in an ill-typed program). Additionally, this is the reason why we abandon the `LENGTH` condition in `CU.Case_of`: if a program has a semantics in $\mathcal{C}$, this means that all the lengths of tuples of arguments of constructors match the number of bound variables in appropriate alternatives. Thus, we do not have to check it on runtime (in practice, this property is guaranteed by the type checking).

5.3 Fake bottom

The $\mathcal{FB}$ language addresses a problem similar to saving environments. In the rule `C.App4`, in the `LEFT` premise we do not use the argument stack (`args`), but then we have to resurrect it in the `RIGHT` premise. We cannot just leave the addresses on the stack, which was the case in the $\mathcal{CU}$ semantics, because the length of the stack determines whether the `App1` or `App2` rule should be used when entering a non-updatable closure. We have to forget about something and then bring it back to life, and our goal is to minimize its size. In an abstract machine, it means minimization of elements on a stack.

As with environments, we make the argument stack global. Instead of forgetting about its elements (or saving on a stack of an abstract machine), we mark a suffix of the stack dead. In an abstract machine we store a pointer to the first dead element, which serves as a bottom of the stack; hence the name *fake bottom*. In the `LEFT` premise of `FB.App4` we emulate an empty stack by setting the fake bottom to the whole length. Doing this we forget its value, but then we have to resurrect only one `nat` (or store one pointer on the stack).

5.3.1 $\mathcal{FB}$ semantics

The $\mathcal{FB}$ language uses the same expressions as $\mathcal{CU}$. It differs only in semantics. First, we introduce the type for fake bottom:

```
Definition fake_bottom := nat.
```

Now, we treat the argument stack as a pair: a stack and its fake bottom:

```
Inductive config : Set :=
| Config : heap → expr → env → (argstack * fake_bottom) →
    addresses → config
| Enter : heap → address → (argstack * fake_bottom) → env →
    config.
```

```
Notation " << a , b , c , d , e >> " := (Config a b c d e)
                                        (at level 70).
Notation " << a , b , c , d >> " := (Enter a b c d)
                                    (at level 70).
```

Since we want to make the stack global, we have to include it in values. We add it to the type of semantics:

```
Inductive Sem : config →
  (value * env * (argstack * fake_bottom)) → Prop :=

| App1 : forall Gamma p args clovars bind e some_clo sigma
                fake_bot real_args fake_args
  (ONHEAP : Gamma p = Some (Lf_n clovars bind e, some_clo))
  (REAL   : real_args = firstn (length args - fake_bot) args)
  (FAKE   : fake_args = skipn (length args - fake_bot) args)
  (LENGTH : bind > length args - fake_bot),
  << Gamma, p, (args, fake_bot), sigma >> ↓
      (Val_pap Gamma p real_args, sigma, (fake_args, fake_bot))

| App2 : forall Gamma p args Value e bind clo clovars sigma
                fake_bot
  (ONHEAP  : Gamma p = Some (Lf_n clovars bind e, clo))
  (LENGTH  : bind <= length args - fake_bot)
  (PREMISE : << Gamma, e, firstn bind args ++ sigma,
                  (skipn bind args, fake_bot), clo >> ↓ Value),
  << Gamma, p, (args, fake_bot), sigma >> ↓ Value

| App4 : forall Gamma p args Value q addrs e clovars clo_e
                Delta sigma theta args0 fake_bot fake_bot0
  (ONHEAP1 : Gamma p = Some (Lf_u clovars 0 e, clo_e))
  (LEFT    : << (clog Gamma p), e, sigma, (args, length args),
                  clo_e >> ↓ (Val_pap Delta q addrs, theta,
                  (args0, fake_bot0)))
  (RIGHT   : << set Delta p (make_pap q addrs), q,
                  (addrs++args0, fake_bot), theta >> ↓ Value),
  << Gamma, p, (args, fake_bot), sigma >> ↓ Value

(...)
where " a ↓ b " := (Sem a b).
```

5.3.2 Certified compiler $\mathcal{CU} \rightarrow \mathcal{FB}$

The compiler to $\mathcal{FB}$ is an identity function:

```
Definition compile (e : CU.expr) : FB.expr := e.
```

The completness theorem which states the equivalence of the $\mathcal{CU}$ and $\mathcal{FB}$ semantics is formulated as follows:

```
Lemma completeness :
forall Config Value tau
  (EVAL : CU.Sem Config (Value, tau)),
  match Config with
  | CU.Config Gamma e sigma args clo ⇒
      FB.Sem (FB.Config Gamma e sigma (args, 0) clo)
             (Value, tau, (nil, 0))
  | CU.Enter Gamma p args sigma ⇒
      FB.Sem (FB.Enter Gamma p (args, 0) sigma)
             (Value, tau, (nil, 0))
  end.
```

5.4 Current closure pointer

When we enter a closure, we take the tuple of addresses from that closure as the C-stack. In practice, it means that we have to copy the whole tuple. This is absolutely inefficient, since those variables are always accessible on the heap, and will not change during evaluation of the closure.

Instead of copying, we store only an address of the closure in a register called *current closure (CC) pointer* and access the addresses indirectly through an offset of that pointer.

5.4.1 $\mathcal{CCP}$ Semantics

The $\mathcal{CCP}$ language also uses the $\mathcal{CU}$ expressions. The semantics depends now on different formalization of the `look_up` and `map_env` functions, since they do not take two-stack environment as their arguments now, but a B-stack (the global environment), a heap and an address to a closure on that heap (instead of a C-stack):

```
Definition look_up (e : env) (h : heap) (p : address) (v : var)
  : option address :=
match v with
| B_ind n ⇒ nth n e
| C_ind n ⇒ match h p with Some (_, clo) ⇒ nth n clo
```

```
                                | None ⇒ None end
end.

Fixpoint map_env (e : env) (h : heap) (a : address) (vs : vars)
  {struct vs} : option addresses := (...)
```

Types of configurations contain an address instead of the whole tuple of addresses:

```
Inductive config : Set :=
| Config : heap → expr → env → (argstack * fake_bottom) →
    address → config
| Enter : heap → address → (argstack * fake_bottom) → env →
    config.
```

The semantics does not differ much from the $\mathcal{FB}$ semantics. The only difference is in the way we call the `look_up` and `map_env` functions and when entering a closure we store its address, not copy its tuple of addresses:

```
Inductive Sem : config →
  (value * env * (argstack * fake_bottom)) → Prop :=

| Cons : forall Gamma C xs sigma addrs cleanup args fake_bot
                pclo
  (MAPENV  : map_env sigma Gamma pclo xs = Some addrs)
  (ARGSNIL : fake_bot = length args),
  << Gamma, Constr C xs cleanup, sigma, (args, fake_bot),
       pclo >> ↓ (Val_con Gamma C addrs, (skipn cleanup sigma),
       (args, fake_bot))

| App2 : forall Gamma p args Value e bind clo clovars sigma
                fake_bot
  (ONHEAP  : Gamma p = Some (Lf_n clovars bind e, clo))
  (LENGTH  : bind <= length args - fake_bot)
  (PREMISE : << Gamma, e, firstn bind args ++ sigma,
                (skipn bind args, fake_bot), p >> ↓ Value),
  << Gamma, p, (args, fake_bot), sigma >> ↓ Value

(...)
```

5.4.2 Certified compiler $\mathcal{FB} \rightarrow \mathcal{CCP}$

As may be expected, the compiler is an identity function:

```
Definition compile (e : FB.expr) : CCP.expr := e.
```

If we are in a closure, its address should be a value of the CC pointer. But there is a problem at the very beginning: at the moment we start the evaluation of the outermost expression, the heap is empty and we are not in any closure, and thus we do not have a value for the pointer. The solution is that it can point anywhere, because there should be no C-indices in the outermost expression. We can generalize it to: whenever a C-stack in the $\mathcal{FB}$ semantics is empty, there should be no C-indices in the expression, and the pointer may have an arbitrary value. It makes the formulation of the general correctness theorem a little cumbersome:

```
Lemma invariant :
forall Config Value
  (EVAL : FB.Sem Config Value),
  match Config with
  | FB.Config Gamma e sigma args clo ⇒
    match clo with
    | nil ⇒ forall p,
      CCP.Sem (CCP.Config Gamma e sigma args p) Value
    | (_::_) ⇒ forall p
      (PCLO : Gamma p = Some (Black_hole, clo)
           ∨ exists clovars, exists bind, exists pe,
              Gamma p = Some (Lf_n clovars bind pe, clo)),
      CCP.Sem (CCP.Config Gamma e sigma args p) Value
    end
  | FB.Enter Gamma p args sigma ⇒
      CCP.Sem (CCP.Enter Gamma p args sigma) Value
  end.
```

In the case of initial configurations this theorem simplifies to:

```
Lemma correctness :
forall e Value,
  FB.Sem (FB.Config empty_heap e nil (nil, 0) nil) Value →
  CCP.Sem (CCP.Config empty_heap e nil (nil, 0) 0) Value.
```

5.5 D-CPS ($\mathcal{CCP} \to \mathcal{D}$)

The $\mathcal{D}$ language is the $\mathcal{CCP}$ semantics after the D-CPS transformation. The `E` and `A` constructors are not just flags now, but contain configurations and values respectively:

```
Definition config := CCP.config.
Definition value := (CU.value * env *
                     (argstack * FB.fake_bottom))%type.

Inductive action : Set :=
| E : config → action
| A : value → action.
```

Types of elements of the continuation stack are very promising:

```
Inductive stack_elem : Set :=
| K_alt : alts → FB.fake_bottom → stack_elem
| K_upd : address → FB.fake_bottom → stack_elem.
```

As we may see, `K_upd` has the constant size of 2 pointers[4], and `K_alt` contains a list of alternatives and one pointer. The list of alternatives will be replaced with a pointer to a code store in the next semantics.

Also, the type of the semantics is as follows:

```
Definition stack := list stack_elem.
Inductive Sem : (action * stack) → value → Prop := (...)
```

We skip the detailed description of the semantics here, because it is created by the already presented method. The theorems that state completeness of the machine are as follows (where `D.App_Eval` is a rule form the $\mathcal{D}$ semantics, and `ARGSNIL` is a name of a premise of another rule):

```
Lemma completeness_lemma :
forall Config Val0
  (CCPS : CCP.Sem Config Val0)
  Val1 K
  (DS : D.Sem (A Val0, K) Val1),
  D.Sem (E Config, K) Val1.
Proof with intros; eauto.
intros ? ? ?.
induction CCPS...
eapply D.App_Eval; try rewrite <- ARGSNIL...
Qed.
```

[4] Coq uses a unary representation of natural numbers, so the size is not really constant. Of course, the type `nat` only models a real address or a machine integer type, and so the `K_upd` constructor models a stack frame of a constant size.

```
Lemma completeness :
forall Config Value
  (CCPS : CCP.Sem Config Value),
  D.Sem (E Config, nil) Value.
Proof with eauto.
intros; eapply completeness_lemma...
Qed.
```

The shortness of those proofs may serve as an another argument for the usefulness of the method of the D-CPS transformation.

5.6 Virtual machine

The previous semantics describe an abstract machine. It uses expressions as a part of its state, stores lambda-forms on the heap and remember lists of alternatives on the continuation stack. Additionally, we create new expressions during updates.

In this section we present a virtual machine which has its own set of instructions. A program is held as a store holding lists of instructions. The store is constant, it cannot change during evaluation of the program. Instead of expressions, alternatives and lambda-forms, only pointers to those elements are now present in the configuration.

The compilation to the $\mathcal{VM}$ code-store procedure is certifying.

5.6.1 Code pointers and instructions

There is a special type for code-pointers. In later implementations it can be easily replaced, for example with `nat`, because we do not match against pointers inside the kind `Set` (except for equality, but it does not pose a problem). So far, it is useful in proofs of completeness and in understanding of updates in the $\mathcal{VM}$ semantics.

```
Inductive code_pointer : Set := (...)
```

The idea behind the set of instructions is based on an observation that during evaluation all the expressions, lambda-forms and alternatives stored in elements of the state are subexpressions of the source term or are created during updates. Now, we concentrate on the former, the latter will be dealt with in the next section.

In the Coq memory model, we may see a Coq expression representing a $\mathcal{D}$ term as a tree of separate memory cells, each holding one node of the expression, joined by pointers. We make some of the pointers explicit, so, instead of storing a subexpression, we store a pointer in the state (see Figure 5.10).

We point to three kinds of entities: expressions (each alternative points to its body), list of alternatives (we will put a pointer to such list on the continuation stack when eval-

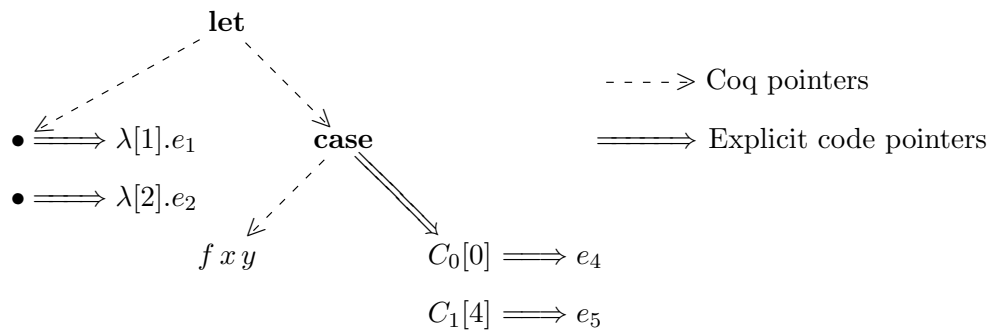
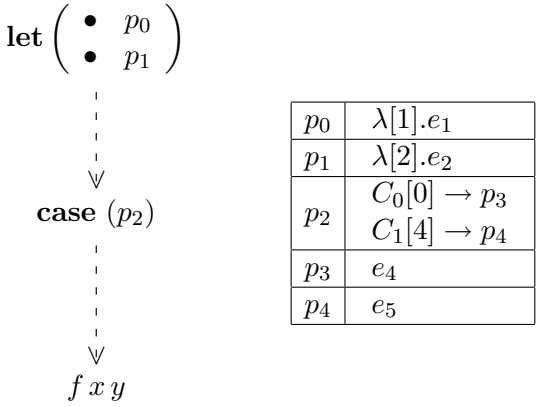


Figure 5.10: Coq pointers and explicit code pointers

uating a case expression), lambda-forms (let expressions contain a list of such pointers, and we will allocate such pointers, instead of lambda-forms, on the heap).

The difference between expressions and instructions is that elements of the latter are linear, that is each has at most one recursive argument, which is the tail of the list of instructions. The last element of each list is always an application or a constructor. Other recursive arguments were substituted with code pointers pointing to a code-store (see Figure 5.11).



p_0	$\lambda[1].e_1$
p_1	$\lambda[2].e_2$
p_2	$C_0[0] \to p_3$ $C_1[4] \to p_4$
p_3	e_4
p_4	e_5

Figure 5.11: A list of instructions and a code-store

The definition of the type of lists of instructions is very similar to the language of expressions:

```
Inductive expr : Set :=
| App : var → vars → CU.cleanup → expr
| Constr : constructor → vars → CU.cleanup → expr
| Letrec : list (vars * code_pointer) → expr → expr
| Case : expr → code_pointer → expr.

Definition lambda_form := gen_lambda_form (Expr := expr).
Definition alt := gen_alt (Expr := code_pointer).
Definition alts := list alt.
```

The virtual machine runs along the list and perform each instruction. If it is `Letrec`, it creates and allocates the closures on the heap (it needs a pair consisting of a pointer to the lambda-form and a list of its free variables), and then continue with evaluation of the 'in' expression. If it is `Case`, the pointer to a list of alternatives is stored on the continuation stack, and we continue with evaluation of the expression inside the case expression.

In the code-store, there are instructions for expressions, lambda-forms and alternatives. Hence the type (the `CS_None` constructor should not be created by a compilation procedure):

```
Inductive cs_node : Set :=
| CS_expr : expr → cs_node
| CS_alts : alts → cs_node
| CS_lf : lambda_form → cs_node
| CS_None.

Definition code_store := list (code_pointer * cs_node).
```

The code-store is then an association list. The procedure `find_cs` allows finding appropriate elements. It returns `CS_None` if there is no element stored at the pointer:

```
Fixpoint find_cs (cs : code_store) (cp : code_pointer) :
  cs_node := (...)
```

5.6.2 $\mathcal{VM}$ semantics

The `code_pointer` type has three constructors:

```
Inductive code_pointer : Set :=
| CP_user : nat → code_pointer
| CP_pap : nat → code_pointer
| CP_cons : constructor → nat → code_pointer.
```

The constructor `CP_user` is supposed to point to a place in the code-store where an element that corresponds with a subexpression of the outermost expression is stored. In turn, `CP_pap` and `CP_cons` are supposed to point to a place where code that represents expressions that may be generated by updates procedures are held.

Except for an expression, the virtual machine holds the current list of instructions. That is why the types of configurations are very similar to those of the $\mathcal{D}$ language. Note that now a closure is a pair consisting of a code-pointer (to a lambda-form) and a tuple of addresses:

```
Definition closure := (code_pointer * addresses)%type.
Definition closures := list closure.
Definition heap := address → option closure.

Inductive config : Set :=
| Config : heap → expr → env → (argstack * FB.fake_bottom) →
    address → config
| Enter : heap → address → argstack * FB.fake_bottom → env →
    config.

Inductive ret_value : Set :=
| Val_con : heap → constructor → addresses → ret_value
| Val_pap : heap → address → addresses → ret_value.

Definition value :=
  (ret_value * env * (argstack * FB.fake_bottom))%type.
```

The semantics takes additional argument, the code-store. The type of predicate defining the machine is as follows:

```
Reserved Notation " cs @@ a ↖ b " (at level 70,
                                   no associativity).

Inductive Sem (cs : code_store) : (action * stack) → value →
  Prop :=
(...)
where "cs @@ a ↖ b " := (Sem cs a b).
```

The semantics operate similarly to the $\mathcal{D}$ semantics, but it sometimes has to search for a new list of instructions in the code-store. For example, rules for case expressions are as follows (recall the type of `Case`, the constructor argument `p_als` is a pointer):

```
| Case_of_Eval : forall Gamma e p_als sigma pclo Value args
                          fake_bot K
  (PREMISE : cs @@ (E (<< Gamma, e, sigma, (args, length args),
                pclo >>), K_alt p_als fake_bot :: K) ↖ Value),
  cs @@ (E (<< Gamma, Case e p_als, sigma, (args, fake_bot),
    pclo >>), K) ↖ Value

| Case_of_Apply : forall p_als als pclo Value bind Delta p_e0
                          e0 C addrs theta args0 fake_bot
                          fake_bot0 K
  (CS1      : find_cs cs p_als = CS_alts als)
  (SELECT   : nth C als = Some (Alt bind p_e0))
  (CS2      : find_cs cs p_e0 = CS_expr e0)
  (PREMISE : cs @@ (E (<< Delta, e0, addrs++theta,
                (args0, fake_bot), pclo >>), K) ↖ Value),
  cs @@ (A (Val_con Delta C addrs, theta, (args0, fake_bot0)),
    K_alt p_als fake_bot :: K) ↖ Value
```

When evaluating the `Case` instruction we create a new frame on the continuation stack containing a pointer to the list of alternatives, and continue with the next instruction. When we have the value, we look at the pointer on the stack, find the list of alternatives in the code-store and choose the right alternative. Each alternative contains a number of bound variables and a pointer to its body. We adjust the environment, and look for the body inside the code-store. The body is the new current list of instructions.

The update rules assume that in the code-store there are appropriate lists of instructions under the addresses `CP_pap` and `CP_cons`. The address `CP_pap n` should point to a lambda-form with zero arguments equivalent to application of a variable to `n` (just like the ones created by `make_pap` function). Similarly, `CP_cons C n` is a constructor `C` with `n` arguments (equivalent to `make_cons` function).

```
| App3 : forall p C addrs Delta theta args fake_bot
         fake_bot_dummy K Value
  (ARGSNIL : fake_bot = length args)
  (PREMISE : cs @@ (A (Val_con
               (set Delta p (CP_cons C (length addrs), addrs))
               C addrs, theta, (args, fake_bot)), K) ↖ Value),
  cs @@ (A (Val_con Delta C addrs, theta, (args, fake_bot)),
    K_upd p fake_bot_dummy :: K) ↖ Value

| App4 : forall p Value q addrs Delta theta args0 fake_bot
                   fake_bot0 K
```

```
  (PREMISE : cs @@ (E
        (<< set Delta p (CP_pap (length addrs), q :: addrs),
        q, (addrs++args0, fake_bot), theta >>), K) ↖ Value),
  cs @@ (A (Val_pap Delta q addrs, theta, (args0, fake_bot0)),
    K_upd p fake_bot :: K) ↖ Value
```

5.6.3 Certifying compiler $\mathcal{D} \to \mathcal{VM}$

The compilation procedure consists of three steps:

1. Flattening of an expression into a code-store,
2. Adding instructions representing expressions that may be generated during updates,
3. Verifying whether the code-store is equivalent to the expression.

The last step is certifying, that is we have a proof in Coq that if the verifier responds positively, the compiler output is equivalent to the input.

Flattening expressions into code-store

The idea of flattening procedure is straightforward. It recursively walks the structure of expressions, isolating subexpressions. Pointers are created as successive natural numbers, using an additional counter given as an argument. It creates a code-store and an initial list of instructions.

The implementation, thought, looks a little more complicated than that, which is a result of the restricted Coq `Fixpoint` construction syntax. For example, it does not allow us to define flattening of lambda-forms and alternatives as separate mutually recursive functions.

```
Fixpoint flatten (n : nat) (e : D.expr) {struct e}
  : nat * code_store * expr :=
match e with
| CU.App x xs cu ⇒ (n, nil, App x xs cu)
| CU.Constr C xs cu ⇒ (n, nil, Constr C xs cu)
| CU.Letrec lfs e ⇒
    match fold_right (* Flattening definitions *)
    (fun lf res ⇒ match lf, res with
      | Lf pi clovars bind e, (n1, cs1, vars_ptrs) ⇒
        match flatten n1 e with
        | (n2, cs2, fe2) ⇒ (S n2,
```

```
            (CP_user n2, CS_lf (Lf pi clovars bind fe2))
               :: cs2 ++ cs1,
            (clovars, CP_user n2) :: vars_ptrs)
        end
      end)
    (n, nil, nil) lfs with
    | (n3, cs3, vars_ptrs) ⇒
      match flatten n3 e with (* Flattening the expression *)
      | (n0, cs, fe) ⇒ (n0, cs++cs3, Letrec vars_ptrs fe)
      end
    end
| CU.Case e als ⇒
    match fold_right (* Flattening alternatives *)
    (fun al res ⇒ match al, res with
      | Alt bind e0, (n0, cs0, als0) ⇒
        match flatten n0 e0 with
        | (n1, cs1, fe1) ⇒ (S n1,
          (CP_user n1, CS_expr fe1) :: cs1 ++ cs0,
          Alt bind (CP_user n1) :: als0)
        end
      end)
    (n, nil, nil) als with
    | (n2, cs2, als2) ⇒
      match flatten n2 e with (* Flattening the expression *)
      | (n3, cs3, fe) ⇒ (S n3, (CP_user n3, CS_alts als2)
            :: cs2 ++ cs3,
        Case fe (CP_user n3))
      end
    end
end.
```

Instructions for updates

In the $\mathcal{D}$ semantics, updates create lambda-forms from the `Val_pap` and `Val_cons` values. Those values, in turn, are created from constructor and application expressions. In $\mathcal{VM}$, it is then sufficient to create instructions for updated closures for each constructor and each possible partial application in the input expression (all expressions created by updates use only C indices, so only one lambda-form for each constructor and a number of arguments of possible application is needed).

We introduce a new datatype which holds information about constructors with a

number of their arguments, and a number of maximal number of arguments of the lambda-forms:

```
Definition app_stat := nat.
Definition cons_stat := ListSet.set (constructor * nat).
Definition statistics := (app_stat * cons_stat)%type.
```

We define a predicate `UB` (for 'upper bound'), which holds if a statistics holds information about all the needed data of a $\mathcal{VM}$ list of instructions (or more):

```
Inductive UB (* upper bound *) : statistics → expr → Prop :=
| UB_App : forall stat x xs cu,
  UB stat (App x xs cu)
| UB_Cons : forall stat_app stat_cons C xs cu
  (INSTAT : set_In (C, length xs) stat_cons),
  UB (stat_app, stat_cons) (Constr C xs cu)
| UB_Letrec : forall stat lfs e
  (UBEXPR : UB stat e),
  UB stat (Letrec lfs e)
| UB_Case : forall stat cp e
  (UBEXPR : UB stat e),
  UB stat (Case e cp).
```

We extend this definition to lambda-forms, code-store nodes, code stores, and whole configurations:

```
Inductive UB_lf : statistics → lambda_form → Prop := (...)
Inductive UB_csn : statistics → cs_node → Prop := (...)
Inductive UB_cs : statistics → code_store → Prop := (...)
Inductive UB_conf : statistics → action * stack → Prop := (...)
```

Now, we can define a strongly specified function to gather statistics from a list of instructions. Its type guarantees that the result is an upper bound of the input expression. We extend it to code-store nodes and code stores.

```
Definition gather_stats : forall e, {stat | UB stat e}.
Proof with intros; simpl in *; auto.
induction e.
exists (length v0, nil)...
exists (0, (c, length v) :: nil); constructor...
destruct IHe as [stat UBe]; exists stat...
destruct IHe as [stat UBe]; exists stat...
Qed.
```

```
Definition gather_stats_csn : forall csn,
                                {stat | UB_csn stat csn}.
Definition gather_stats_cs : forall cs, {stat | UB_cs stat cs}.
```

We can generate $\mathcal{VM}$ instructions from the statistics using the `impl_for` functions:

```
Fixpoint impl_for_pap (n : app_stat) : code_store :=
match n with
| O ⇒ (CP_pap 0, CS_lf (Lf Dont_update nil 0 (App (C_ind 0)
          (map C_ind (from 1 0)) 0))) :: nil
| S m ⇒ (CP_pap n, CS_lf (Lf Dont_update nil 0 (App (C_ind 0)
             (map C_ind (from 1 n)) 0))) :: impl_for_pap m
end.

Fixpoint impl_for_cons (xs : cons_stat) : code_store :=
match xs with
| (C, n) :: xs0 ⇒ (CP_cons C n, CS_lf
       (Lf Dont_update nil 0
          (Constr C (map C_ind (from 0 n)) 0)))
       :: impl_for_cons xs0
| nil ⇒ nil
end.
```

Verification predicates

The central point of the verification procedure are two families of predicates: `Rebuild` and `Implements`.

The former states that lists of instructions (alternatives, lambda-forms, heaps, configurations, values, action, stacks, closures) are equivalent to $\mathcal{D}$ expressions (etc.). That is, if we substitute code-pointers with normal Coq pointers, we will get $\mathcal{D}$ expressions:

```
Inductive Rebuild (cs : code_store) : expr → D.expr → Prop :=
| Reb_App : forall x xs cu,
  Rebuild cs (App x xs (cu)) (CU.App x xs (cu))
| Reb_Cons : forall C xs cu,
  Rebuild cs (Constr C xs (cu)) (CU.Constr C xs (cu))
| Reb_Letrec : forall lfs lfs0 e e_r
  (REXPR : Rebuild cs e e_r)
  (RLFS  : Rebuild_lfs cs lfs lfs0),
  Rebuild cs (Letrec lfs e) (CU.Letrec lfs0 e_r)
| Reb_Case : forall e e_r p_als als als_r
  (REXPR   : Rebuild cs e e_r)
```

```
  (FINDALS : find_cs cs p_als = CS_alts als)
  (RALS    : Rebuild_alts cs als als_r),
  Rebuild cs (Case e p_als) (CU.Case e_r als_r)
with Rebuild_lfs (cs : code_store) : list (vars * code_pointer)
  → list D.lambda_form → Prop := (...)
with Rebuild_alts (cs : code_store) : alts → D.alts → Prop :=
  (...)
Inductive Rebuild_lf (cs : code_store) : lambda_form →
  D.lambda_form → Prop := (...)
Inductive Heap_similar (cs : code_store) : D.heap → heap →
  Prop := (...)
Inductive Config_similar (cs : code_store) : D.config →
  config → Prop := (...)
Inductive Value_similar (cs : code_store) : D.value → value →
  Prop := (...)
Inductive Action_similar (cs : code_store) : D.action →
  action → Prop := (...)
Inductive Stack_similar (cs : code_store) : D.stack → stack →
  Prop := (...)
Inductive Similar (cs : code_store) : D.action * D.stack →
  action * stack → Prop := (...)
```

The `Implements` predicate state that a code-store has all the code needed to implement statistics. For example, if a statistics has constructor C with n arguments as its element, appropriate $\mathcal{VM}$ instruction is present in the code-store under the address `CP_cons C n`:

```
Inductive Implements : code_store → statistics → Prop :=
| Impl : forall stat_app stat_cons cs
  (APP  : forall n
          (LEQ : n <= stat_app),
          find_cs cs (CP_pap n) = CS_lf
            (Lf Dont_update nil 0 (App (C_ind 0)
            (map C_ind (from 1 n)) 0)))
  (CONS : forall C n
          (INSTAT : set_In (C, n) stat_cons),
          find_cs cs (CP_cons C n) = CS_lf
            (Lf Dont_update nil 0
              (Constr C (map C_ind (from 0 n)) 0))),
  Implements cs (stat_app, stat_cons).
```

Compiler and Verifier

The overview of the compiler is shown in Figure 5.12.

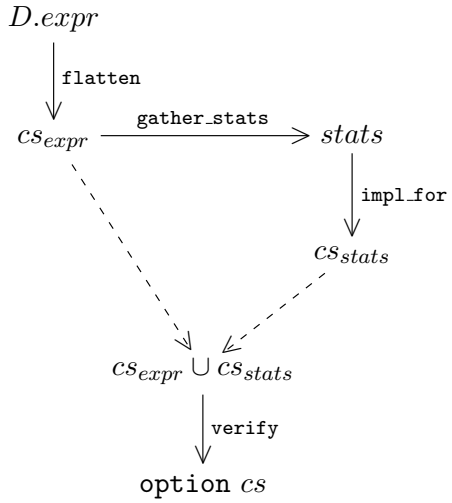


Figure 5.12: $\mathcal{D}$ to $\mathcal{VM}$ compiler overview

The main theorem used by the compiler is formulated as follows. It says that if a code store implements its own statistics and it rebuilds to an expression (the `SIMILAR` condition), their semantics are equivalent:

```
Lemma invariant :
forall cs DConf DVal stat
  (UB_IMPL : Implements cs stat)
  (UB_CS   : UB_cs stat cs)
  (DEVAL   : D.Sem DConf DVal)
  Conf
  (UB_CONF : UB_conf stat Conf)
  (SIMILAR : Similar cs DConf Conf),
  exists Val, Sem cs Conf Val ∧ Value_similar cs DVal Val.
```

Therefore, the first step to verify a result of compilation is to check whether the `Rebuild` predicate applies:

```
Definition verify : forall (e : D.expr) (cs : code_store)
  (e_fla : expr), {Rebuild cs e_fla e} + {~ Rebuild cs e_fla e}.
```

The compiler flatten the expression and checks the `Implements` predicate:

```
Definition compile (d_e : D.expr) : option
  { x : code_store * expr * statistics
  | match x with
    (cs, e, stat) ⇒
      UB stat e ∧
      UB_cs stat cs ∧
      Implements cs stat ∧
      Rebuild cs e d_e
    end }.
```

5.6.4 Extraction of the compiler

The compilation procedure from $\mathcal{D}$ to $\mathcal{VM}$ is certifying, that is whenever the verifier says **true**, we are sure that the compiler output is semantically equivalent to the input. The job of flattening procedure is very simple, but an implementation without a proof of correctness is always prone to mistakes. That is why it is advisable to test the compiler and see whether it fails for any input programs.

To do that we use the Coq mechanism of program extraction. It enables an automatic translation of the Coq definitions into a general-purpose functional programming language like Haskell or OCaml.

We use the Coq command `Extraction Language Haskell` to tell the Coq system that we want the extracted program to be in Haskell, and then `Recursive Extraction` on the `VM.compile` function. The result is about 800 lines of code. Of course the code does not preserve the rich Coq types, for example the extracted `compile` function has the following type:

```
compile :: Expr0 → Option (Prod (Prod Code_store Expr1)
                                Statistics)
```

Coq exports all its datatypes (including tuples and `Option`), but this type is clearly isomorphic to Haskell's:

```
compile :: Expr0 → Maybe (Code_store, Expr1, Statistics)
```

Though the function is not strongly specified in Haskell, the extraction procedure guarantees that the semantics of the definition in Coq is the same as the semantics of the extracted Haskell program.

The extracted code can be found in the `vm/extraction` directory. The file `Core.hs` is the exported compiler (we decorated the datatypes with the `deriving(Show)` Haskell directive and fixed the incorrect indentation provided by the extractor). The file `User.hs` has some example terms, which can be flatten and verified using the `c` function, for example:

```
Prelude> :l User
Ok, modules loaded: User, Core.
*User> c let2
```

5.7 Alternative implementation of environments

The whole derivation of the virtual machine revolves around flattening expressions and the environments management. There latter may be done in several different ways, and our solution with the environment stack is different than in GHC or a virtual machine ISTG proposed by Encina and Peña [12]. While GHC manages environments in a very advanced way using concrete architecture memory models, the ISTG machine is at about the same level of abstraction as our virtual machine.

The difference is that ISTG uses one stack, which represents the environment, the continuation stack and the argument stack. This way we do not have to copy arguments from the argument stack to the environment stack. On the other hand, to put some arguments on the stack, we need to use the old environment, so we cannot remove it first. Thus, we put the arguments on top of environment, which is going to be cleared up the next moment, so cleaning procedure is supposed to remove elements from the middle of the stack, which requires sliding. Figure 5.13 presents an example where e evaluates to $C_0\, a_1\, a_2\, a_3$. The evaluation of $B3\, B2\, B3$ puts the third and fourth element of the current stack on top of the stack. We then enter the closure under the address stored in the fourth element of the stack. Because we enter another closure, we need to clean up the old environment build by evaluation of the old closure.

To compare the two solutions (ours and ISTG's), we need to take two issues into account:

- Our solution needs to copy arguments from one stack to another, and Encina and Peña's solution needs to slide the stack. The complexity of the latter is linear to the number of elements of the prefix before the slid fragment, which happens to be new arguments. Thus, the complexity of both is exactly the same.

- Though it does not affect the efficiency, in a real-life compiler we sometimes prefer to use as little stacks as possible, because it minimizes the blocks of contiguous memory that needs to be managed.

Both approaches are thus comparable. We choose the separate environment stack, because we want our machine to be low-level, but keep some aspects abstract for possible further optimizations. For example, GHC holds some parts of environments in additional registers or as an offset from the top of the heap. Our approach may be easier adjusted

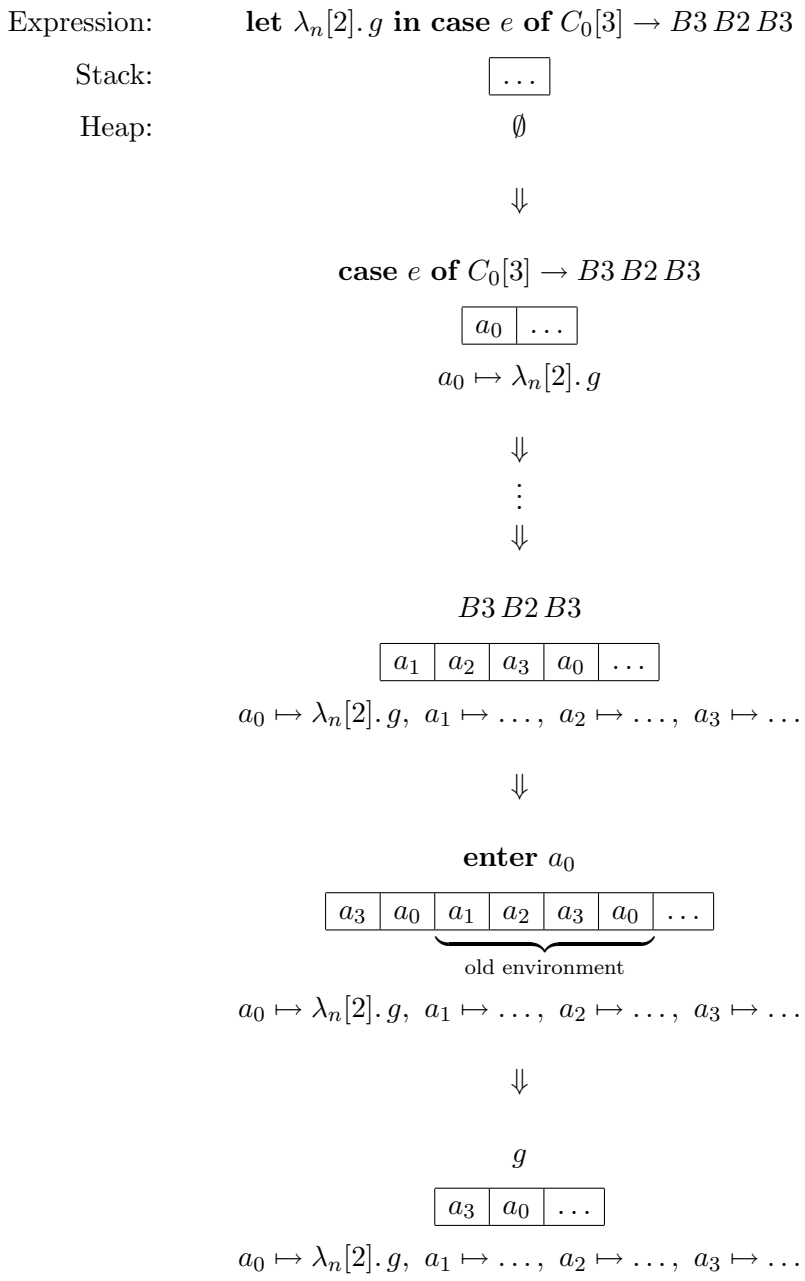


Figure 5.13: Using the argument stack as a global environment.

to such more advanced environment handling, for example by introducing another types of variables (instead of only B and C).

Summary in Polish
Podsumowanie po polsku

Praca ta odnosi się do zagadnienia bezpiecznej kompilacji leniwych języków programowania, takich jak Haskell, Miranda czy Clean. Poprzez bezpieczeństwo rozumiemy semantyczną równoważność wysokopoziomowego wejścia i niskopoziomowego wyjścia kompilatora. Pewność równoważności otrzymujemy dzięki formalnemu matematycznemu dowodowi poprawności, który został mechanicznie zweryfikowany przez system dowodzenia, taki jak Coq, Hol czy Twelf. O programie, dla którego podano taki dowód, mówimy, że jest certyfikowany.

Semantyka leniwych języków programowania jest bardzo odległa od modelu obliczeń współczesnych komputerów. Leniwa ewaluacja oferuje wiele możliwości (takich jak na przykład ko-indukcyjne, a więc potencjalnie nieskończone, struktury danych), których nawet bardzo nieefektywna kompilacja nie jest trywialna. Jest to największa przeszkoda w konstrukcji kompilatora, która staje się jeszcze bardziej uciążliwa w przypadku kompilatorów certyfikowanych. W tej pracy przedstawiamy pierwszy certyfikowany kompilator dla języków leniwych, który tłumaczy wyrażenia wysokopoziomowego języka funkcyjnego na instrukcje maszyny abstrakcyjnej. Chociaż nie jest to pełny kompilator do asemblera czy niskopoziomowego języka w rodzaju C, nasza maszyna jest w pełni imperatywna, a więc dalsza implementacja pełnego kompilatora nie musi już odnosić się do zawiłej semantyki języków leniwych, a potrzebuje jedynie generatora kodu dla konkretnej architektury i systemu automatycznego zarządzania pamięcią.

Naszym centralnym punktem odniesienia jest semantyka dla znormalizowanych wyrażeń języka Haskell, którą zaproponował Peyton Jones i Salkild – Spineless Tagless G-machine (STG). W pracy omawiamy cztery kroki, które prowadzą nas do certyfikowanego kompilatora:

- Proponujemy semantykę naturalną dla znormalizowanych wyrażeń.
- Adaptujemy do naszych potrzeb metodę transformacji ewaluatorów do implementacji maszyn abstrakcyjnych zaproponowaną przez Danvy'ego i innych. Dzięki tej metodzie udowadniamy równoważność naszej semantyki naturalnej z maszyną STG.
- Zastanawiamy się, co to znaczy, że kompilator jest poprawny i szukamy minimalnego (a więc najłatwiejszego do zrealizowania i udowodnienia) zabioru założeń, który taki kompilator musi spełniać.

- Przedstawiamy naszą implementację, która rozpoczyna od języka znormalizowanych wyrażeń, a kończy na instrukcjach maszyny wirtualnej. Poprawność wszystkich kroków kompilacji udowodniono w systemie dowodzenia twierdzeń Coq.

Praca ta jest zaledwie opisem projektu w systemie Coq, który liczy sobie około 11 tyś. wierszy kodu (250 definicji i 330 twierdzeń). Wysoki poziom skomplikowania reguł poszczególnych semantyk sprawia, że dowody na papierze byłyby bardzo zawiłe, a ich późniejsza weryfikacja byłaby prawie niemożliwa. Coq wymusza na nas bardzo precyzyjne sformalizowanie najdrobniejszych szczegółów, ale za to gwarantuje poprawność dowodów.

Bibliography

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Miller [29], pages 8–19.

[2] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 233–246, San Francisco, California, January 1995. ACM Press.

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

[4] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15, New York, NY, USA, 2008. ACM.

[5] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-machine. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 244–258, New York, NY, USA, 1988. ACM.

[6] Adam Chlipala. A verified compiler for an impure functional language. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–106, New York, NY, USA, 2010. ACM.

[7] Clean homepage: `http://clean.cs.ru.nl`.

[8] Coq homepage: `http://coq.inria.fr`.

[9] Olivier Danvy. Defunctionalized interpreters for programming languages. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 131–142, New York, NY, USA, 2008. ACM.

[10] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press.

[11] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *LPAR'07: Proceedings of the 14th international conference on Logic for programming, artificial intelligence and reasoning*, pages 211–225, Berlin, Heidelberg, 2007. Springer-Verlag.

[12] Alberto de la Encina and Ricardo Peña. Formally deriving an STG machine. In Miller [29], pages 102–112.

[13] Alberto de la Encina and Ricardo Peña. Proving the correctness of the STG machine. In Ricardo Pena and Thomas Arts, editors, *IFL*, volume 2670 of *Lecture Notes in Computer Science*, pages 88–104. Springer, 2003.

[14] Alberto de la Encina and Ricardo Peña. From natural semantics to C: A formal derivation of two STG machines. *Journal of Functional Programming*, 19(1):47–94, 2009.

[15] John Fairbairn and Stuart Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 34–45, London, UK, 1987. Springer-Verlag.

[16] Stephan Frank, Martin Grabmüller, Petra Hofstedt, Dirk Kleeblatt, Technische Universität Berlin, Pierre R. Mai, and Stefan alexander Schneider. Safety of compilers and translation techniques- status quo of technology and science.

[17] Sabine Glesner. Optimierende Compiler: Vertrauen ist gut, Verifikation ist besser! Technical report, Fakultät für Informatik (Fak. f. Informatik), Institut für Programmstrukturen und Datenorganisation (IPD), 2005. Interner Bericht. Fakultät für Informatik, Universität Karlsruhe; 2005,28, ISSN: 1432-7864.

[18] Joshua Guttman, Vipin Swarup, and John Ramsdell. VLISP: A verified implementation of Scheme. In *Lisp and Symbolic Computation*, pages 8–1, 1995.

[19] Haskell homepage: `http://www.haskell.org`.

[20] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. *Higher Order Symbol. Comput.*, 22(1):3–66, 2009.

[21] HOL homepage: `http://hol.sourceforge.net`.

[22] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag.

[23] Thomas Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Not.*, 39(4):125–138, 2004.

[24] John Launchbury. A natural semantics for lazy evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, January 1993. ACM Press.

[25] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM.

[26] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[27] Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.*, 16(4-5):415–449, 2006.

[28] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. pages 33–41. American Mathematical Society, 1967.

[29] Dale Miller, editor. *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, Uppsala, Sweden, August 2003. ACM Press.

[30] Miranda homepage: `http://miranda.org.uk`.

[31] Jon Mountjoy. The spineless tagless G-machine, naturally. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 34, No. 1, pages 163–173, Baltimore, Maryland, September 1998. ACM Press.

[32] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 33(5):333–344, 1998.

[33] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[34] Simon L. Peyton Jones and David Lester. *Implementing functional languages: a tutorial*. Prentice-Hall International, 1992.

[35] Simon L. Peyton Jones and Jon Salkild. The spineless tagless G-machine. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 184–201, London, England, September 1989. ACM Press.

[36] Maciej Piróg and Dariusz Biernacki. A systematic derivation of the STG machine verified in Coq, 2010. Submitted for publication.

[37] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [38].

[38] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[39] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[40] Twelf homepage: `http://twelf.plparty.org`.