Type Systems for Closure Conversions

John Hannan

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802 USA

Abstract. We consider the problem of analyzing and proving correct simple closure conversion strategies for a higher-order functional language. We specify the conversions as deductive systems, making use of annotated types to provide constraints which guide the construction of the closures. We exploit the ability of deductive systems to specify concisely complex relationships between source terms and closure-converted terms. The resulting specifications and proofs are relatively clear and straightforward. The use of deductive systems is central to our work as we can subsequently encode these systems in the LF type theory and then code them in the Elf programming language. The correctness proofs can also be coded in this language, providing machine-checked versions of these proofs.

1 Introduction

Closure conversion is the process of transforming functions containing free variables into a closures, a representation of a function that consists of a piece of code for the function and a record containing the free variables occurring in the original function. This process consists not only of converting functions to closures but also of replacing function calls with the invocation of the code component of closures on the actual parameter and the closure itself (which will contain values for the free variables). Closure conversion is a critical step in the compilation of higher-order functional languages, and different closure conversion strategies can have remarkably different run-time behaviors in terms of space utilization. Reasoning about these conversions can become complicated as the conversion themselves become more complicated. We believe that a means for analyzing various conversion strategies will provide a useful tool for understanding and correctly implementing closure conversion.

1.1 Contribution

The main contribution of this paper is the development of type systems to specify and prove correct various closure conversion strategies. In particular, the type systems are reasonably simple and clearly express the relationship between source terms and closure converted terms. We specify the conversions as deductive systems axiomatizing judgments which relate expressions containing functions and those containing closures. These systems make critical use of annotated types to provide constraints which guide the construction of the closures. The use of deductive systems is critical to this work, as we subsequently encode these systems into the LF [7] type theory and then the Elf programming language [11], providing both experimental implementations of closure conversion but also machine-checked proofs of correctness. In the current paper we focus only on the deductive systems, but most of the systems presented here have been implemented and proved correct in Elf. We include only the deductive systems and statements of the relevant correctness theorems. For the full proofs and the Elf code implementing the closure conversions and specifying the proofs see the full version of the paper, available as a technical report from our institution.

The kinds of closure conversions addressed in this paper are simple, but the methods developed demonstrate the capabilities of type systems for describing closure conversions. Recent work on space efficient closure representation has demonstrated the efficiency possible if closures are carefully constructed using a variety of information [12]. While we have not yet considered such advanced closure conversion representations, we believe that our approach will provide a useful tool for reasoning about and proving correct such techniques.

1.2 Related Work

The problem of correctness for closure conversion has recently been addressed in [14]. The approach used in that work includes a flow analysis to generate constraints which ensures that the closure conversion algorithm generates closures that consistently use the correct procedure calling protocol in the presence of multiple calling protocols (for example, one protocol for use with closures as procedures and one for use with λ-abstractions as procedures). Based on techniques from abstract interpretation, their approach requires the introduction of an abstract notion of terms and evaluation (their "occurrence evaluator") and the relationship between their original language and this abstract version. Annotations are then added to provide constraint information and they prove that their conversion satisfies these constraints. Their proof of correctness, however, only shows what we called soundness in [6]: if the source program evaluates, then the converted term does too. (Wand proved the converse using different techniques in [13].) Our initial motivation was to demonstrate how equivalent results could be produced using type systems.

The idea of using type systems to specify constraints of programs and to guide the translation of programs has been successfully used by Tofte and Talpin to describe region inference for Standard ML programs. Region inference detects blocks of storage that can be allocated and deallocated in a stack-like fashion. Their use of annotated types has motivated some of our techniques for annotating function types with information regarding the free variables required to call the function.

Related work on compiler correctness includes [3] where compiler optimizations based on strictness analysis are proved correct. This work, however, considers CPS translations and definitions that resemble denotational semantics.

1.3 Organization of Paper

The remainder of the paper is organized as follows. In Sec. 2 we introduce a basic closure conversion specification and a verification of its correctness. In Sec. 3 we extend the basic conversion to a selective one and demonstrate how its correctness is a direct generalization of the basic case. In Sec. 4 we extend the selective conversion to one in which not all free variables need be included in a closure. Finally in Sec. 5 we conclude by mentioning some additional conversion strategies and our intent to verify them. In the appendix we give a brief introduction to the LF type theory and its application to specifying deductive systems.

2 Simple Closure Conversion

We begin by considering a simple closure conversion specification in which every function is converted into a closure.

2.1 Source and Target Languages

We consider just the simply typed λ-calculus as the source language:

$$E ::= x \mid \lambda x.E \mid E \mathbin{@} E$$

in which $E_1 \mathbin{@} E_2$ represents application. For our first presentation of closure conversion types play no role, and so we can also consider this method as applying to an untyped language. But in subsequent sections we rely heavily on types and a typed language.

The target language of closures consists of the following:

$$\begin{aligned} M &::= x \mid n\#C \mid C \mid M \mathbin{@_c} M \\ C &::= c \mid [\lambda c.\lambda x.M, L] \\ L &::= \cdot \mid L, M \end{aligned}$$

in which $E_1 \mathbin{@_c} E_2$ represents application in which the value of E_1 will be a closure. The meta-variables M, C and L range over terms in the target language, closures (and closure variables), and lists of target terms, respectively. The λ-abstraction of the source language is replaced by the closure construction $[\lambda c.\lambda y.M, L]$ in which the bound variable c corresponds to the closure itself, the bound variable y corresponds to the bound variable of the λ-abstraction, M corresponds to the body of the λ-abstraction and L is a list of the free variables of the λ-abstraction. (We refer to c as the *closure-bound variable* of the closure.) We include the bound variable c to approximate the structure of closures as described in [2] in which a closure is invoked by fetching the first field of the closure and applying it to its arguments including the closure itself.

We represent variables in two ways: locally bound variables (i.e., variables bound by the nearest enclosing lambda abstraction in a source term) are represented as in the source language; non-locally bound variables are represented by the term $n\#C$ in which n is a positive integer (de Bruijn index) and C is either a closure or a closure-bound variable. For example, the term $\lambda x.\lambda y.(x\,y)$ is represented in the target language by the term

$$[\lambda c_1.\lambda x.([\lambda c_2.\lambda y.((1\#c_2)\,y), (\cdot, x)]),\ \cdot]$$

For both the source and target languages we can provide an operational semantics, each given by a set of inference rules which can be directly represented as LF signatures. Both semantics implement call-by-value to weak-head normal form. For the source language we introduce the judgment $e \hookrightarrow_s v$ and axiomatize it via the following two rules:

$$\frac{}{\lambda x.E \hookrightarrow_s \lambda x.E}$$

$$\frac{E_1 \hookrightarrow_s \lambda x.E' \qquad E_2 \hookrightarrow_s V_2 \qquad E'[V_2/x] \hookrightarrow_s V}{E_1 \,@\, E_2 \hookrightarrow_s V}$$

For the target language we introduce the judgment $e \hookrightarrow_t v$ and axiomatize it via the following rules:

$$\frac{}{[\lambda c.\lambda y.M, L] \hookrightarrow_t [\lambda c.\lambda y.M, L]}$$

$$\frac{M_1 \hookrightarrow_t [\lambda c.\lambda y.M', L] \quad M_2 \hookrightarrow_t V_2' \quad M'[V_2'/y][[\lambda c.\lambda y.M', L]/c] \hookrightarrow_t V'}{M_1 \,@_c\, M_2 \hookrightarrow_t V'}$$

$$\frac{nth\ N\ L\ V'}{N\#\,[\lambda c.\lambda y.M, L] \hookrightarrow_t V'}$$

The judgment $(nth\ N\ L\ V)$ expresses the relation that the N^{th} element in the list L is V.

This specification of evaluation using closures focuses on the access of free variables in function bodies, but not on the mechanism by which values are stored into the record component of the closure. In the specification above, the substitution $M'[V_2/y]$ replaces all occurrences of y in M' with value V_2. As y may occur in the record component of a closure, this substitution can have the effect of loading the value V into any number of closures. While this is hardly realistic, this convention makes the specification particularly simple and easy to analyze. Our future goal is to provide further refinements of this specification to reflect more accurately the manipulation of closures.

2.2 The Closure Conversion Specification

To specify closure conversion we could have introduced a type system for source terms in which the type of a function explicitly provides information about the shape of the desired closure for the function. Then the translation from source to target languages would be trivial. We can, in fact, combine these two operations (typing and translating) into a single deductive system, in which types play a reduced role due to the presence of contextual information.

We specify closure conversion as a translation from source to target languages. We introduce the judgment $E \Longrightarrow M$ which denotes the property that source term E closure converts to term M. As we need some additional information to specify the conversion, we also introduce the judgment $\langle L, x, c\rangle \rhd E \Longrightarrow M; L'$ in which L and L' are lists of target language terms (typically variables), x is a source language term (typically a variable), c is a closure (or closure variable), E is a source term and M is a target term. The judgment can be read as follows: L is the list of variables (*or terms substituted for these variables*) that occur free and must therefore be included in an enclosing closure; x is the bound variable of the nearest enclosing λ-abstraction; c is the closure variable of the nearest enclosing closure (to be constructed); E is the source term to be converted; M is the converted term; and L' is the (possible) extension of L which includes the free variables of M. Note that the first judgment $E \Longrightarrow M$ is really just a special case of the second judgment where the sets L and L' are empty. We prefer to use two distinct judgments as it simplifies and clarifies the correctness proofs. Note also that while x and y in rules (1.2) and (1.4) denote variables in the source and target languages, respectively, in the remaining rules occurrences of x, y, z and c, though suggestive of variables, can range over arbitrary terms of the appropriate syntactic class (source term, target term or closure).

Finally, we need a third judgment, $L \rhd x \mapsto N; L'$ which is used to generate a de Bruijn index N for a variable x. This judgment relies on the property that bound variables are distinct in the given source term. The judgment can be understood as follows: starting with (target) variable list L, source variable x closure converts to the N^{th} variable of L'. The list L' is different from L if only if the variable y to which x converts is not already in L. In this case, y is added to the end of L, creating L'. The complete system for basic closure conversion is given in Fig. 1.

The first two rules specify the top-level conversion. In rule (1.2) the universal quantification of the variables x, y, and c ensures that these variables are arbitrary (and do not already occur in any assumptions). (In our implementation in Elf this property is automatically maintained from our use of higher-order syntax and Π-quantification of these variables.) Note that the variable x may occur free in E and the variables y and c may occur free in M. The variables are bound by the universal quantifiers in the antecedent of the rule and are bound by λ-abstractions in the conclusion. This manipulation of variables, motivated by the higher-order syntax used in our implementation, eliminates any need for variable conventions or renaming. We use implication to introduce the hypothesis $x \Longrightarrow y$ (an instance of the judgment $E \Longrightarrow M$), instead of maintaining an explicit context of information. Again, our implementation supports this operation and structuring the rule in this way simplifies the correctness proofs. The operation $L + L'$ denotes the new list of variables obtained by appending to L those elements of L' (in order) not already occurring in L.

The structure of these rules, in particular, the use of universal quantification and implication is critical when we encode these rules into an LF signature and Elf program and apply the Propositions-as-Types analogy. Then, for example, the judgment $\forall x \forall y \forall c (x \Longrightarrow y \supset \langle \cdot, x, c \rangle \rhd E \Longrightarrow M; L)$, when viewed as a type denotes a functional type taking four arguments (terms for x, y, and c, and an object of type $x \Longrightarrow y$). Thus a deduction of this judgment, when viewed as an object, represents a function which when applied to terms E', M', C' and an object representing the deduction $E' \Longrightarrow M'$, yields an object representing a deduction of $\langle \cdot, E', C' \rangle \rhd E[E'/x] \Longrightarrow M[M'/y, C'/c]; L$.

2.3 Verifying the Conversion

We are now in a position to state the correctness criteria for our closure conversion specification. First we state two lemmas about the auxiliary judgments $L \rhd x \mapsto N; L'$ and $\langle L, x, c \rangle \rhd E \Longrightarrow M; L'$:

Lemma 1. *For all lists L, L', source term E, target term M and natural number N, if $\vdash L \rhd E \mapsto N; L'$ and $\vdash$ (nth N L' M) then $\vdash E \Longrightarrow M$.*

The proof is by induction on the structure of the deductions for the judgment $L \rhd E \mapsto N; L'$.

Lemma 2. *1. For all source terms E, V, x, target term M, lists L, L', and closure term (or variable) C, if $\vdash E \hookrightarrow_s V$ and $\vdash \langle L, x, C \rangle \rhd E \Longrightarrow M; L'$ then there exists a V' such that $\vdash M \hookrightarrow_t V'$ and $\vdash V \Longrightarrow V'$;*

2. For all source terms E, x, target terms M, V', lists L, L', and closure term (or variable) C, if $\vdash \langle L, x, C \rangle \rhd E \Longrightarrow M; L'$ and $\vdash M \hookrightarrow_t V'$ then there exists a V such that $\vdash E \hookrightarrow_s V$ and $\vdash V \Longrightarrow V'$.

The proof is straightforward by induction on the structure of the deduction for $E \hookrightarrow_s V$ and then by cases on the structure of the deduction for $\langle L, x, c \rangle \rhd E \Longrightarrow M, L'$.

Finally, we can state the main theorem.

Theorem 3. *1. For all source terms E, V and target term M, if $\vdash E \hookrightarrow_s V$ and $\vdash E \Longrightarrow M$ then there exists a V' such that $\vdash M \hookrightarrow_t V'$ and $\vdash V \Longrightarrow V'$;*

2. For all source term E and target terms M, V', if $\vdash E \Longrightarrow M$ and $\vdash M \hookrightarrow_t V'$ then there exists a V such that $\vdash E \hookrightarrow_s V$ and $\vdash V \Longrightarrow V'$.

This kind of correctness statement is in the same spirit as one of the correctness statements for the compiler found in our earlier work on compiler correctness [6].

$$\frac{E_1 \Longrightarrow M_1 \qquad E_2 \Longrightarrow M_2}{(E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2)} \tag{1.1}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y \;\supset\; \langle \cdot, x, c\rangle \rhd E \Longrightarrow M; L)}{\lambda x.E \Longrightarrow [\lambda c.\lambda y.M, L]} \tag{1.2}$$

$$\frac{\langle L, x, c\rangle \rhd E_1 \Longrightarrow M_1; L' \qquad \langle L', x, c\rangle \rhd E_2 \Longrightarrow M_2; L''}{\langle L, x, c\rangle \rhd (E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2); L''} \tag{1.3}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y \;\supset\; \langle \cdot, x, c\rangle \rhd E \Longrightarrow M; L')}{\langle L, x', c'\rangle \rhd \lambda x.E \Longrightarrow [\lambda c.\lambda y.M, L']\,;\, L + L'} \tag{1.4}$$

$$\frac{x \Longrightarrow y}{\langle L, x, c\rangle \rhd x \Longrightarrow y; L} \tag{1.5}$$

$$\frac{L \rhd z \mapsto N, L'}{\langle L, x, c\rangle \rhd z \Longrightarrow (N\#c); L'} \;\; (z \neq x) \tag{1.6}$$

$$\frac{x \Longrightarrow y}{\cdot \rhd x \mapsto 1; (\cdot, y)} \tag{1.7}$$

$$\frac{x \Longrightarrow y}{(L, y) \rhd x \mapsto 1; (L, y)} \tag{1.8}$$

$$\frac{L \rhd x \mapsto N; L'}{(L, y) \rhd x \mapsto N{+}1; (L', y)} \tag{1.9}$$

Fig. 1. Basic Closure Conversion

3 Selective Closure Conversion

As pointed out in [14], avoiding closure creation plays an important role in generating efficient code for higher-order languages. In the basic closure conversion specification of the previous section, all source language functions were converted to closures. We consider here the possibility of selective closure conversion in which source language functions are not converted to closures, but rather left as functions. Our focus in this section is not so much on determining exactly when closures need not be converted, but rather on demonstrating that both closures and functions can be handled together, ensuring that applications in the target language are constructed with the proper procedure calling conventions.

The approach taken in [14] uses a relatively complex flow analysis, related to some abstract interpretation techniques. Using a type system, we provide a straightforward account of selective conversion. Instead of explicit constraints we have types, and type inference provides the means for resolving constraints imposed by these types. For demonstration purposes, we consider only one case in which a source language function is not translated into a closure: when the function contains no free variables.

3.1 Extending the Languages

We begin by considering the types for our languages and by extending the target language. The types consist of some collection of base types and two kinds of function types:

$$\tau ::= a \mid \tau{\rightarrow}\tau \mid \tau{\rightarrow_c}\tau$$

A λ-abstraction will be given type $\tau_1{\rightarrow}\tau_2$ if it should not be closure converted. A λ-abstraction will be given type $\tau_1{\rightarrow_c}\tau_2$ if it should be closure converted.

The target language is extended by including λ-abstractions and a second form of application:

$$M ::= x \mid n\#c \mid \ [\lambda c.\lambda x.M, L] \mid M \mathbin{@_c} M \mid \lambda x.M \mid M\,M$$

We extend the operational semantics for the language with the two rules:

$$\frac{}{\lambda y.M \hookrightarrow_t \lambda y.M}$$

$$\frac{M_1 \hookrightarrow_t \lambda y.M \qquad M_2 \hookrightarrow_t V_2 \qquad M[V_2/y] \hookrightarrow_t V}{M_1\,M_2 \hookrightarrow_t V}$$

3.2 Adding Selective Conversion

The closure conversion specification is modified in a few ways. First, we include the source language type in the judgments $E \Longrightarrow M$, $\langle L, x, c\rangle \rhd E \Longrightarrow M; L'$, and $L \rhd x \mapsto N; L'$. The judgments become $E \Longrightarrow M : \tau$, $\langle L, x, c\rangle \rhd E \Longrightarrow M : \tau; L'$, and $L \rhd x \mapsto N : \tau; L'$. Second, the original rules for translating λ-abstractions and applications use the function type $\tau_1{\rightarrow_c}\tau_2$ to indicate that the λ-abstraction is converted to a closure and the application contains an operator that should evaluate to a closure. Finally, we add two new rules for when a λ-abstraction does not convert to a closure and the corresponding rules for the new application. The complete system is given in Fig. 2.

The rules for converting applications differ only in the type given to the operator. The new rules (2.3) and (2.7) for converting λ-abstractions differ from the original ones (in which closures are created) by requiring that the list of free variables occurring in the term be empty. This is enforced by the occurrence of '$\cdot$' on the right-hand sides of the antecedents in these two new rules.

To ensure that this conversion produces terms that make proper use of closures we have the following consistency lemma.

$$\frac{E_1 \Longrightarrow M_1 : (\tau_1 \to \tau_2) \qquad E_2 \Longrightarrow M_2 : \tau_1}{(E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@} M_2) : \tau_2} \tag{2.1}$$

$$\frac{E_1 \Longrightarrow M_1 : (\tau_1 \to_c \tau_2) \qquad E_2 \Longrightarrow M_2 : \tau_1}{(E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2) : \tau_2} \tag{2.2}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y : \tau_1 \ \supset \ \langle \cdot, x, c \rangle \rhd E \Longrightarrow M : \tau_2; \cdot)}{\lambda x.E \Longrightarrow \lambda y.M : (\tau_1 \to \tau_2)} \tag{2.3}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y : \tau_1 \ \supset \ \langle \cdot, x, c \rangle \rhd E \Longrightarrow M : \tau_2; L)}{\lambda x.E \Longrightarrow [\lambda c.\lambda y.M, L] : (\tau_1 \to_c \tau_2)} \tag{2.4}$$

$$\frac{\langle L, x, c \rangle \rhd E_1 \Longrightarrow M_1 : (\tau_1 \to \tau_2); L' \quad \langle L', x, c \rangle \rhd E_2 \Longrightarrow M_2 : \tau_1; L''}{\langle L, x, c \rangle \rhd (E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@} M_2) : \tau_2; L''} \tag{2.5}$$

$$\frac{\langle L, x, c \rangle \rhd E_1 \Longrightarrow M_1 : (\tau_1 \to_c \tau_2); L' \quad \langle L', x, c \rangle \rhd E_2 \Longrightarrow M_2 : \tau_1; L''}{\langle L, x, c \rangle \rhd (E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2) : \tau_2; L''} \tag{2.6}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y : \tau_1 \ \supset \ \langle \cdot, x, c \rangle \rhd E \Longrightarrow M : \tau_1; \cdot)}{\langle L, x', c' \rangle \rhd \lambda x.E \Longrightarrow \lambda y.M : (\tau_1 \to \tau_2); L} \tag{2.7}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y : \tau_1 \ \supset \ \langle \cdot, x, c \rangle \rhd E \Longrightarrow M : \tau_1; L')}{\langle L, x', c' \rangle \rhd \lambda x.E \Longrightarrow [\lambda c.\lambda y.M, L'] : (\tau_1 \to_c \tau_2); \, L + L'} \tag{2.8}$$

$$\frac{x \Longrightarrow y : \tau}{\langle L, x, c \rangle \rhd x \Longrightarrow y : \tau; L} \tag{2.9}$$

$$\frac{L, z \mapsto N : \tau, L'}{\langle L, x, c \rangle \rhd z \Longrightarrow (N \# c) : \tau; L'} \quad (z \neq x) \tag{2.10}$$

$$\frac{x \Longrightarrow y : \tau}{\cdot \rhd x \mapsto 1 : \tau; (\cdot, y)} \tag{2.11}$$

$$\frac{x \Longrightarrow y : \tau}{(L, y) \rhd x \mapsto 1 : \tau; (L, y)} \tag{2.12}$$

$$\frac{L \rhd x \mapsto N : \tau; L'}{(L, y) \rhd x \mapsto (N{+}1) : \tau; (L', y)} \tag{2.13}$$

Fig. 2. Selective Closure Conversion

Lemma 4. *For all source term E and target term M,*

1. *if* $\vdash E \Longrightarrow M : \tau_1 \rightarrow \tau_2$ *and* $M \hookrightarrow_t V'$ *then* $V' = \lambda y.M'$ *for some* M'*;*
2. *if* $\vdash E \Longrightarrow M : \tau_1 \rightarrow_c \tau_2$ *and* $M \hookrightarrow_t V'$ *then* $V' = [\lambda c.\lambda y.M']$ *for some* M'*.*

The proof is straightforward by induction of the structure of deductions. The fact that closed λ-abstractions need not be converted to closures is obvious and this lemma ensures that we can selectively convert λ-abstractions and still have correct procedure call protocols.

3.3 Verifying the Conversion

Adapting the proof of correctness for basic closure conversion to selective closure conversion is straightforward and the Elf program representing the proof is only slightly longer than the proof for basic conversion.

Theorem 5. *1. For all source terms* E, V *and target term* M*, if* $\vdash E \hookrightarrow_s V$ *and* $\vdash E \Longrightarrow M : \tau$ *then there exists a* V' *such that* $\vdash M \hookrightarrow_t V'$ *and* $\vdash V \Longrightarrow V' : \tau$*;*

2. For all source term E *and target terms* M, V'*, if* $\vdash E \Longrightarrow M : \tau$ *and* $\vdash M \hookrightarrow_t V'$ *then there exists a* V *such that* $\vdash E \hookrightarrow_s V$ *and* $\vdash V \Longrightarrow V' : \tau$*.*

4 Lightweight Closure Conversion

The point of a closure is to provide a function body with access to non-local variables at the time the function is called. In particular the function call which originally bound some of these variables may have returned by the time this closure is accessed. If, however, certain variables can be shown only to be accessed during the evaluation of the function body which bound them, then these variables do not necessarily need to be included as part of a closure, as their bindings will be available elsewhere. We can exploit this idea and reduce the number of variables included in a closure, possibly eliminating the need for a closure entirely in some cases. Detecting when this is possible requires detailed analysis of the expression being closure converted.

In related work we have developed a static escape analysis for λ-terms [5]. This analysis, presented as a type system, determines when a bound variable can escape its scope at run time, i.e., when the variable may be accessed even after the function in which it was bound has returned. This situation occurs when bound variables occur inside of function bodies and these functions can be returned as values of the function for which the variable is a formal parameter. An example illustrates this relatively simple idea. Consider the function $\lambda x.\lambda f.f\,x$. If this function is applied to some value v, then the result of the function call will be the closure consisting of the function $\lambda f.f\,x$ and the binding $x \mapsto v$. In this case, x is the bound variable of a function, but this variable continues to exist after returning from the function.

A variable simply occurring in the body of another function is not a sufficient condition to imply that it escapes. Consider the term $\lambda x.((\lambda f.f\,x)(\lambda y.y))$. By some simple observations of this term, we can see that the occurrence of x in the body will not escape its scope.

Our analysis uses a judgment of the form $\Gamma \rhd E : \tau \Rightarrow E^s$ in which Γ is a type context, E is a source term, τ is an annotated type, and E^s is an annotated term. The annotated target language is a typed λ-calculus, but it includes two forms for each construct in the source language: one regular form and one annotated form.

$$M ::= x \mid x^s \mid \lambda x.M \mid \lambda^s x^s.M \mid M \mathbin{@} M \mid M \mathbin{@^s} M$$

A term of the form $\lambda^s x^s.M$ indicates that the variable x^s cannot escape from its scope. The term $M \mathbin{@^s} N$ indicates that the value of the term M will be a function whose bound variable cannot escape its scope.

The analysis essentially determines which lambda abstractions and applications in the source term can be annotated in the target term. For example, the two term given above could be annotated as

$$\lambda x.\lambda f^s.f^s\,x \ \text{ and } \ \lambda x^s.((\lambda f^s.f^s\,x^s)(\lambda y^s.y^s)).$$

The annotations in the first term indicate that f^s cannot escape its scope but that x may. The annotations in the second term indicate that no variables can escape their scope. In [5] we use this information to provide a stack-based implementation of a functional language by first translating it into an annotated form. Annotated variables are allocated space on a run-time stack and can be deallocated when a function returns. Closures are only created at run time.

Applying this analysis to closure conversion we can observe that the only variables required for a function closure are those variables that occur free in the function *and* can escape their scope. If they can escape their scope, then their binding will not necessarily be part of the "global" environment at the time the function body is evaluated. Variables that cannot escape (those annotated after analysis) can be allocated on a stack (because they can be safely deallocated upon returning from the corresponding function call) and their values can always be accessed, when required, from this stack. Closures which do not contain such non-escaping variables are called *lightweight closures* in [14], though they do not focus directly on whether variables escape their scope. They refer to these non-escaping variables as *dynamic variables*.

We can specify such lightweight closure conversion as the composition of our escape analysis and an extended version of the conversion specification given in the previous section. Note that we could also combine these two into a single specification, but for clarity we will keep the two distinct and focus only on the extended version of closure conversion using annotated terms. The specification for lightweight conversion includes the previous rules for selective conversion (Fig. 2) and the additional rules of Fig. 3. The first five rules simply treat annotated λ-abstractions, applications and local variables as before in selective conversion. Rule (3.6) provides the essential difference. In this case, non-local, annotated variables are not included in the set of variables used for constructing closures. The variables are simply translated into the appropriate target language variables. Compare this rule with (2.10).

$$\frac{E_1 \Longrightarrow M_1 : (\tau_1 \rightarrow_c \tau_2) \qquad E_2 \Longrightarrow M_2 : \tau_1}{(E_1 \mathbin{@^s} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2) : \tau_2} \tag{3.1}$$

$$\frac{\forall x^s \forall y \forall c (x^s \Longrightarrow y : \tau_1 \ \supset \ \langle \cdot, x^s, c \rangle \rhd E \Longrightarrow M : \tau_2 ; L')}{\lambda^s x^s . E \Longrightarrow [\lambda c . \lambda y . M, L'] : (\tau_1 \rightarrow_c \tau_2)} \tag{3.2}$$

$$\frac{\langle L, x, c \rangle \rhd E_1 \Longrightarrow M_1 : (\tau_1 \rightarrow_c \tau_2) ; L' \quad \langle L', x, c \rangle \rhd E_2 \Longrightarrow M_2 : \tau_1 ; L''}{\langle L, x, c \rangle \rhd (E_1 \mathbin{@^s} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2) : \tau_2 ; L''} \tag{3.3}$$

$$\frac{\forall x^s \forall y \forall c (x^s \Longrightarrow y : \tau_1 \ \supset \ \langle \cdot, x^s, c \rangle \rhd E \Longrightarrow M : \tau_1 ; L')}{\langle L, x', c' \rangle \rhd \lambda^s x^s . E \Longrightarrow [\lambda c . \lambda y . M, L'] : (\tau_1 \rightarrow_c \tau_2) ; \ L + L'} \tag{3.4}$$

$$\frac{x^s \Longrightarrow y : \tau}{\langle L, x^s, c \rangle \rhd x^s \Longrightarrow y : \tau ; L} \tag{3.5}$$

$$\frac{z^s \Longrightarrow y : \tau}{\langle L, x, c \rangle \rhd z^s \Longrightarrow y : \tau ; L} \ (z \neq x) \tag{3.6}$$

Fig. 3. Lightweight Closure Conversion

When using lightweight closures we do not explicitly pass dynamic variables to functions which need them (as done in [14]). Instead we expect an implementation to exploit the availability of the variables (actually, the values bound to them), located in some global store such as a stack. Some simple calculations allows each function to determine the location at run time of these dynamic variables on the stack [5].

To adequately characterize the correctness of lightweight closure conversion we would need to introduce an operational semantics for our closure language that provides a finer specification of storage than given by the one in Sec. 2. We leave this for future work.

5 Conclusions and Future Work

We have presented a series of closure conversion specifications using type systems. The systems are relatively simple and for basic and selective conversion we have constructed a machine-checked proof of correctness in Elf. A corresponding proof for lightweight conversion is in progress.

We are applying our technique to other strategies for closure conversion, with the expectation of constructing clear specifications for these strategies and also correctness proofs. Our goal is to model the space-efficient closure representations constructed in the Standard ML of New Jersey compiler[12]. An important aspect here is to represent closures in a manner which allows storage to be deallocated or reclaimed as soon as data is no longer needed. Before attempting these more complex closure representations and conversion strategies we need a solid understanding of some of the basic issues and techniques of closure conversion, and a suitable framework for expressing and reasoning about them. The current work provides such initial experience.

A common program transformation, prior to closure conversion is CPS translation which produces λ-terms which closely reflect the control-flow and data-flow operations of a traditional machine architecture. We have specified and proved correct the translation from source program to CPS program using the same approach given here, using deductive systems to specify operational semantics and the CPS translation. The proof is straightforward and captures the essential notion of continuations. We have not, however, combined this translation with closure conversion. This is the subject of future work.

We intend to analyze more detailed translation strategies that incorporate *caller-save* registers and *callee-save* registers as described in [2, 12]. Doing this will require more complex analyses but we expect that using type systems we can adequately express them. We intend to analyze and prove correct the notion of *safe for space* complexity as described in [2]. To accomplish this we will consider a variety of static analyses on programs including lifetime analysis and closure strategy analysis (which determines where to allocate each closure).

References

1. Andrew Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Conf. Rec. of the 16th ACM Symposium on Principles of Programming Languages*, pages 293–302, 1989.
2. Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
3. Geoffrey Burn and Daniel Le Métayer. Proving the correctness of compiler optimisations based on strictness analysis. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Languages Implementation and Logic Programmig*, volume 714 of *Lecture Notes in Computer Science*, pages 346–364. Springer-Verlag, 1993.
4. T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.
5. John Hannan. A type-based analysis for stack allocation in functional languages. Submitted, May 1995.
6. John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418. IEEE Computer Society Press, 1992.

7. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.
8. S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in Elf. In Lars Hallnäs, editor, *Extensions of Logic Programming*, pages 299–344. Springer-Verlag LNCS 596, 1992. A preliminary version is available as Technical Report MPI–I–91–211, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, August 1991.
9. Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE Computer Society Press, June 1989.
10. Frank Pfenning. An implementation of the Elf core language in Standard ML. Available via ftp over the Internet, September 1991. Send mail to elf-request@cs.cmu.edu for further information.
11. Frank Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
12. Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 150–161. ACM, ACM Press, 1994.
13. Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In *Proceedings of the Mathematical Foundations of Programming Semantics '91*, volume 598 of *Lecture Notes in Computer Science*, pages 294–311. Springer-Verlag, 1992.
14. Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *Conf. Rec. 21st ACM Symposium on Principles of Programming Languages*, 1994.

A Overview of LF

We briefly review the LF logical framework [7] as realized in Elf [9, 10, 11]. A tutorial introduction to the Elf core language can be found in [8].

The LF calculus is a three-level calculus for *objects*, *families*, and *kinds*. Families are classified by kinds, and objects are classified by *types*, that is, families of kind Type.

$$\begin{array}{lll} \textit{Kinds} & K & ::= \text{Type} \mid \Pi x{:}A.\ K \\ \textit{Families} & A & ::= a \mid \Pi x{:}A_1.\ A_2 \mid \lambda x{:}A_1.\ A_2 \mid A\,M \\ \textit{Objects} & M & ::= c \mid x \mid \lambda x{:}A.\ M \mid M_1\,M_2 \end{array}$$

Family-level constants are denoted by a, object-level constants by c. We also use the customary abbreviation $A \to B$ and sometimes $B \leftarrow A$ for $\Pi x{:}A.\ B$ when x does not appear free in B. Similarly, $A \to K$ can stand for $\Pi x{:}A.\ K$ when x does not appear free in K. A *signature* declares the kinds of family-level constants a and types of object-level constants c.

The notion of definitional equality we consider here is based on $\beta\eta$-conversion. Type-checking remains decidable (see [4]) and it has the advantage over the original formulation with only β-conversion that every term has an equivalent canonical form.

The Elf programming language provides an operational semantics for LF. This semantics arises from a computational interpretation of types, similar in spirit to the way a computational interpretation of formulas in Horn logic gives rise to Pure Prolog. Due to space limitations, we must refer the reader to [8, 9, 11] for further material on the operational semantics of Elf.

Throughout this paper we have used only deductive systems to present solutions to problems. Each of these systems, however, has a direct encoding as an LF signature (a set of constant declarations), and hence, also an Elf program. In particular, an Elf program corresponding to a verification proof, when type-checked, provides a (mostly) machine-checked version of the proof. For lack of space we have not provided the LF signatures or Elf programs corresponding to the systems given in the paper, but the ability to construct these is a critical aspect of our work. The Elf language provides a powerful tool for experimenting with, and verifying, various analyses.

We give here only a brief description of how the deductive systems and proofs described in the paper can be encoded as LF signatures. From there, the encoding of signatures into Elf programs is a direct translation: each signature item becomes an Elf declaration.

We represent a programming language (that we wish to study) via an abstract syntax consisting of a set of object constants for constructing objects of a particular type. For example, we introduce a type *tm* of source programs and collection of object constants for building objects of type *tm*. We use a higher-order abstract syntax to represent functions and this simplifies the manipulation of programs with bound variables.

We represent judgments such as $e \hookrightarrow v$ via a type family $eval : tm \to tm \to type$. For given objects $e : tm$ and $v : tm$, $(eval\ e\ v)$ is a type.

We represent inference rules as object level constants for constructing objects of types such as $(eval\ e\ v)$. For example, an inference rule

$$\frac{A_1 \quad A_2 \quad A_3}{A_0}$$

would be represented as a constant $c : \Pi x_1 : B_1 \cdots \Pi x_n : B_n(A_1^* \to A_2^* \to A_3^* \to A_0^*)$ in which A_i^* is the representation of judgment A_i as a type and the $x_i : B_i$ are the free variables (implicitly universally quantified) of the inference rule. Using such constants we can construct objects, for example, of type $(eval\ e\ v)$, representing the deduction $e \hookrightarrow v$.

Finally, we represent inductive proofs (with the induction over the structure of deductions) as signatures in which each constant represents a case in the inductive proof. For example, to prove a statement of the form "judgment A is derivable iff judgment B is derivable" then we define a new judgment or type family, for example $thm : A \to B \to type$ to express the relationship between objects of type A and objects of type B. Base cases in the inductive proof translate to axioms (objects of base type) while inductive step cases translate to inference rules (objects of functional type). See [6] for examples of this technique.