J. Functional Programming **8** (3): 239–273, May 1998. Printed in the United Kingdom

A type-based escape analysis for functional languages

JOHN HANNAN

Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802 USA
e-mail: hannan@cse.psu.edu

Abstract

An important issue faced by implementors of higher-order functional programming languages is the allocation and deallocation of storage for variables. The possibility of variables escaping their scope during runtime makes traditional stack allocation inadequate. We consider the problem of detecting when variables in such languages do not escape their scope, and thus can have their bindings allocated in an efficient manner. We use an annotated type system to infer information about the use of variables in a higher-order, strict functional language and combine this system with a translation to an annotated language which explicitly indicates which variables do not escape. The type system uses a notion of annotated types which extends the traditional simple type system with information about the extent of variables. To illustrate the use of this information we define an operational semantics for the annotated language which supports both stack and environment allocation of variable bindings. Only the stack allocated bindings need follow the protocol for stacks: their extent may not exceed their scope. Environment allocated bindings can have any extent, and their allocation has no impact on the stack allocated ones. We prove the analysis and translation correct with respect to this operational semantics by adapting a traditional type consistency proof to our setting. We have encoded the proof into the Elf programming language and typechecked it, providing a partially machine-checked proof.

Capsule Review

Typical implementations of functional languages take one of two approaches to closure allocation. Some allocate all closures on the heap, while others use a mixture of stack allocation and heap allocation. Reducing closure allocation costs through stack allocation may yet prove to be the best way to achieve pay-as-you-go efficiency for higher-order functions, but this approach requires an effective analysis to determine which closures may be stack allocated. This paper presents an extended static type system that addresses this important problem. The paper augments a growing body of work that recognizes the power and simplicity of extended static type systems for inferring and representing the results of program analyses. The author defines a type system with annotations indicating which variables have lifetimes that do not escape their lexical scope. A type-directed transformation uses this information to translate a higher-order functional language to an abstract machine with instructions for manipulating both closures and stacks. The type systems, abstract machines, and translations in the paper are clearly and carefully defined. Appropriate correctness theorems are presented, and these theorems are supported by detailed proofs. This paper is an important step towards more efficient implementations of functional languages.

1 Introduction

A higher-order functional language allows functions, with free variables, as values. This feature presents a significant problem to implementors of these languages: the scope of a variable (a static property) and the extent of the same variable (a dynamic property) may not have a simple correspondence, as found in block-structured languages. This condition prohibits the general use of a stack to allocate space for variable bindings. As a simple example consider the λ-term $\lambda x.\lambda f.f\ x$. To apply this function to some value v, we bind the variable x to v and evaluate the body of the function. In block-structured languages the extent of a formal parameter to a function ends when the function returns. But in this case, the value of the body is simply $\lambda f.f\ x$, where x is bound to v. The reference to x in the body has not been accessed.

Typical approaches to implementing functional languages build a *closure* containing both $\lambda f.f\ x$ and the binding of x to v and return this as the result of the function call. The binding for x is still needed in case we ever apply the function $\lambda f.f\ x$ to an argument. Thus the extent of x continues even after function call return. To support this situation, some implementations of higher-order functional languages allocate all variables on a heap and use garbage collection to deallocate inaccessible structures. While this approach has proven viable, we believe that implementations can profit from an analysis that detects cases in which stack allocation of a variable is safe. The ORBIT compiler (Kranz, 1988) for Scheme performed a first-order escape analysis to provide such information, but it did not provide information for higher-order functions. An important advantage of using stack allocation over heap allocation, besides the automatic deallocation of storage, is the reduction in the size of closures that must be created, either at compile time or run time, to support non-local variable access. By detecting variables that can be stack allocated, we also detect that these variables need not be placed in a closure. At best, we may detect situations in which no closure need be created. Thus our analysis can lead to savings of space (by reducing the size of closures) and also time (by reducing the need to store values in closures at run time). It should be noted, however, that Shao and Appel have demonstrated space-efficient closure representations which reduces the space required for closures (by introducing shared closures) (Shao and Appel, 1994).

We develop a static analysis via a type system to perform an escape analysis in which the type of an expression indicates which *variable bindings* can safely be allocated on a stack. This effort contrasts with the work on region analysis (Tofte and Talpin, 1994) which studies the lifetime of *values* and provides a stack discipline for storage of values. This is admittedly a more difficult problem due to the sharing of data structures which occurs. While it might be safe to allocate a binding $x \mapsto v$ on a stack, it might not be safe to allocate the value v on a stack, if this value can be shared with other bindings. The lifetime of the value can exceed the lifetime of the binding. This situation arises with data structures passed as arguments to functions. Rather than copying the entire structure, an implementation typically passes a pointer to the data structure, resulting in sharing.

A simple distinction which illustrates the essential difference between our work and region analysis is that we study properties of the environment (mapping variables to locations) while region analysis studies properties of the store (mapping locations to values). In the cases where values are small and can be copied (when passed as arguments) then both approaches provide similar information. But when values are large and are passed by reference, then sharing of data structures occurs and region analysis provides information regarding the lifetime of these data structures. However, the lifetime of variable bindings is important in its own right because such information can be utilized to construct efficient representations of function closures (Hannan, 1995). In particular, non-escaping variable bindings might not need to be included into a function closure, resulting in faster closure creation and potentially more efficient variable access.

An important contribution of this work is that we allow for both stack and environment-allocated objects, and the allocation of one variable in an environment does not necessarily preclude the earlier allocation of another variable on a stack. This contrasts with previous work which detected when stack allocation alone was sufficient (Banerjee and Schmidt, 1994), or a strict heap allocation policy was used (Appel, 1992). A second contribution of our work is the construction of a simple abstract machine that exhibits the stack behavior and illustrates the difference between stack and environment allocation. We also introduce an operational semantics that uses both a stack and an environment. A third contribution is our proof which demonstrates the correctness of the analysis and the operational semantics, with respect to a traditional operational semantics. We have encoded this proof into the Elf programming language (Pfenning, 1991), providing a partially machine-checked proof (Ibarra, 1997). This effort revealed minor errors and deficiencies in an earlier version of the analysis and its proof. It also enhanced our understanding of some of the more complicated aspects of this work. We believe that the type system, the abstract machine, the definition of consistency (used in the proof), and the encoded version of the proof provide a useful set of tools for experimenting with, and reasoning about, storage allocation techniques in languages.

The remainder of the paper is organized as follows. In section 2 we introduce the syntax of our source and target languages. We also introduce an operational semantics and an abstract machine for the source language. In section 3 we introduce the static analysis and translation from source to target language. These are given as a single deductive system using annotated types to guide the translation. In section 4 we define an operational semantics and an abstract machine for the annotated target language. The abstract machine utilizes a global variable stack and explicitly pops the stack when a variable's block or scope terminates. In section 5 we prove the equivalence of the source and target operational semantics and state the equivalence of the source and target abstract machines. We prove the consistency of these operational semantics with respect to the static analysis, justifying the use of type annotations. Here, consistency includes the traditional notion of subject reduction but also statements justifying the presence of annotations. We also discuss our experience with developing a machine-checked proof in Elf. In section 6 we discuss a possible refinement of the analysis which provides lifetime information for variables. In section 7 we discuss related work, and finally, in section 8 we conclude.

2 Operational semantics

Both the source and target languages are simple higher-order functional languages with call-by-value parameter passing. The source language consists of the simply typed λ-terms extended with constants, a conditional and a fixed-point operator:

$$e \quad ::= \quad c \mid x \mid \lambda x.e \mid e \mathbin{@} e \mid \mathsf{if}\ e\ e\ e \mid \mu f.\lambda x.e$$

where c ranges over some set of constants and $(e_1 \mathbin{@} e_2)$ represents the application of e_1 to e_2. The fixed-point operator μ provides for recursive function definitions. Types are implicit, à la Curry, and terms can be typed via a standard typing system using judgments of the form $\Gamma \rhd e : \tau$. In the next section we introduce a type system which combines this notion of typing with a translation to the target language. We assume Barendregt's 'variable convention' (Barendregt, 1984) in which all bound variables are chosen to be distinct from each other and from the free variables.

The target language is again an extended typed λ-calculus, but it includes two forms of variables and applications: one regular form and one annotated form:

$$\begin{aligned} z, h \quad &::= \quad x \mid x^* \\ m \quad &::= \quad c \mid z \mid \lambda z.m \mid m \mathbin{@} m \mid m \mathbin{@}^* m \mid \mathsf{if}\ m\ m\ m \mid \mu h.\lambda z.m \end{aligned}$$

We use h to range over recursive function names and z to range over other variable names. We assume the same set of constants in both languages. We also assume a one-to-one correspondence between the variables of the source language and the unannotated variables of the target language, and a one-to-one correspondence between the annotated and unannotated variables of the target language. The term x^* will be used to indicate that the value binding for the variable does not escape the scope of x, similarly for recursive function names f^*. The term $(m_1 \mathbin{@}^* m_2)$ will be used to indicate that the value of the term m_1 (if it exists) will be a function whose argument binding does not escape its scope.

We provide an operational semantics for the source language via a set of inference rules specifying call-by-value evaluation to weak-head normal form. We introduce the judgment $\rho \rhd e \hookrightarrow v$, in which ρ is an environment, e is a source language expression, and v is a source language value. We axiomatize it via the rules of Fig. 1. For this deductive system, and others introduced subsequently, we write $\Pi :: \rho \rhd e \hookrightarrow v$ to indicate that Π is a deduction of the judgment $\rho \rhd e \hookrightarrow v$. Values are either constants or function closures ($[\rho, \lambda x.e]$ and $[\rho, \mu f.\lambda x.e]$) consisting of a function and an environment ρ.

We assume an environment is an ordered sequence of bindings (associating distinct variables to values), with the most recent binding on the right end. Thus $\rho\{x \mapsto v\}$ represents the environment obtained after adding the binding of x to v to the environment ρ (with x not already bound in ρ). We use '$\bullet$' to represent the empty environment. Let $\mathsf{dom}(\rho)$ denote the domain of ρ. We write $\rho(x)$ to denote the value bound to x in ρ. (We also use these same constructors and definitions for representing type contexts and variable stacks in later sections.) As a convenience we introduce the notion of prefix for environments and other structures. An environment ρ' is a prefix of ρ, written $\rho' \sqsubseteq \rho$, if either $\rho' = \rho$, or $\rho = \rho''\{x \mapsto v\}$ for some x and v, and

$$\frac{}{\rho \rhd c \hookrightarrow c} \tag{1.1}$$

$$\frac{\rho(x) = v}{\rho \rhd x \hookrightarrow v} \tag{1.2}$$

$$\frac{\rho \rhd e_1 \hookrightarrow \mathsf{true} \quad \rho \rhd e_2 \hookrightarrow v}{\rho \rhd \mathsf{if}\ e_1\ e_2\ e_3 \hookrightarrow v} \qquad \frac{\rho \rhd e_1 \hookrightarrow \mathsf{false} \quad \rho \rhd e_3 \hookrightarrow v}{\rho \rhd \mathsf{if}\ e_1\ e_2\ e_3 \hookrightarrow v} \tag{1.3, 1.4}$$

$$\frac{}{\rho \rhd \lambda x.e \hookrightarrow [\rho, \lambda x.e]} \qquad \frac{}{\rho \rhd \mu f.\lambda x.e \hookrightarrow [\rho, \mu f.\lambda x.e]} \tag{1.5, 1.6}$$

$$\frac{\rho \rhd e_1 \hookrightarrow [\rho', \lambda x.e'] \qquad \rho \rhd e_2 \hookrightarrow v_2 \qquad \rho'\{x \mapsto v_2\} \rhd e' \hookrightarrow v}{\rho \rhd e_1 @ e_2 \hookrightarrow v} \tag{1.7}$$

$$\frac{\begin{array}{c}\rho \rhd e_1 \hookrightarrow [\rho', \mu f.\lambda x.e'] \qquad \rho \rhd e_2 \hookrightarrow v_2 \\ \rho'\{f \mapsto [\rho', \mu f.\lambda x.e']\}\{x \mapsto v_2\} \rhd e' \hookrightarrow v\end{array}}{\rho \rhd e_1 @ e_2 \hookrightarrow v} \tag{1.8}$$

Fig. 1. Source language operational semantics.

ρ' is a prefix of ρ''. An environment can be considered as an abstraction of a heap which contains only variable bindings. We assume that we manipulate only closed terms or terms whose free variables are bound in a given environment.

As an alternative to the operational semantics of Fig. 1, we can also define an abstract machine which implements the source language. An abstract machine provides an intermediate level description of a language by defining a set of rewrite rules that operate on a machine state. Each rule describes a single reduction in the machine state. For our call-by-value source language we have previously defined the CLS machine (Hannan and Miller, 1992), our variant of the SECD machine, which is given in Fig. 2. In this machine, a state is a triple $\langle C, L, S\rangle$ in which C is a sequence of instructions (terms to be reduced), L is a sequence of environments, and S is a stack for holding intermediate results. The machine operates by stepping through a sequence of states with the effect of evaluating the first term on the instruction list C and leaving its value on top of the stack S. The term is evaluated using the first environment in L. The new instructions ap and branch apply a function to its evaluated argument and select a branch of a conditional, respectively. To evaluate a term e with respect to environment ρ we start the machine in the state $\langle e{::}\mathsf{nil}, \rho{::}\mathsf{nil}, \mathsf{nil}\rangle$. If e has a value (with respect to ρ) then the machine will reach a final state $\langle \mathsf{nil}, \mathsf{nil}, v{::}\mathsf{nil}\rangle$ from which the value v can be extracted. In previous work (Hannan and Miller, 1992) we have demonstrated the precise relationship between the operational semantics and this abstract machine, which can simply be expressed as:

$$\rho \rhd e \hookrightarrow v \quad \text{iff} \quad \langle e{::}\mathsf{nil}, \rho{::}\mathsf{nil}, \mathsf{nil}\rangle \Rightarrow \langle \mathsf{nil}, \mathsf{nil}, v{::}\mathsf{nil}\rangle$$

in which all the free variables of e are in $\mathsf{dom}(\rho)$. The original version of this machine described in (Hannan and Miller, 1992) uses a syntax with de Bruijn indices for variables instead of symbolic names. We prefer to use symbolic names here to simplify the presentation.

$$
\begin{array}{lcl}
\langle c{::}C,\ \rho{::}L,\ S\rangle & \Rightarrow & \langle C,\ L,\ c{::}S\rangle \\
\langle x{::}C,\ \rho{::}L,\ S\rangle & \Rightarrow & \langle C,\ L,\ \rho(x){::}S\rangle \\
\langle(\mathsf{if}\ e_1\ e_2\ e_3){::}C, \rho{::}L,\ S\rangle & \Rightarrow & \langle e_1{::}\mathsf{branch}(e_2,e_3){::}C,\ \rho{::}\rho{::}L,\ S\rangle \\
\langle\mathsf{branch}(e_2,e_3){::}C,\ \rho{::}L,\ \mathsf{true}{::}S\rangle & \Rightarrow & \langle e_2{::}C,\ \rho{::}L,\ S\rangle \\
\langle\mathsf{branch}(e_2,e_3){::}C,\ \rho{::}L,\ \mathsf{false}{::}S\rangle & \Rightarrow & \langle e_3{::}C,\ \rho{::}L,\ S\rangle \\
\langle(\lambda x.e){::}C,\ \rho{::}L,\ S\rangle & \Rightarrow & \langle C,\ L,\ \ [\rho,\lambda x.e]\ {::}S\rangle \\
\langle(\mu f.\lambda x.e){::}C,\ \rho{::}L,\ S\rangle & \Rightarrow & \langle C,\ L,\ [\rho,\mu f.\lambda x.e]\ {::}S\rangle \\
\langle(e_1 \mathbin{@} e_2){::}C, \rho{::}L,\ S\rangle & \Rightarrow & \langle e_2{::}e_1{::}\mathsf{ap}{::}C,\ \rho{::}\rho{::}L,\ S\rangle \\
\langle\mathsf{ap}{::}C,\ L,\ [\rho,\lambda x.e]\ {::}v{::}S\rangle & \Rightarrow & \langle e{::}C,\ (\rho\{x{\mapsto}v\}){::}L,\ S\rangle \\
\langle\mathsf{ap}{::}C,\ L,\ [\rho,\mu f.\lambda x.e]\ {::}v{::}S\rangle & \Rightarrow & \langle e{::}C,\ (\rho\{f \mapsto [\rho,\mu f.\lambda x.e]\}\{x{\mapsto}v\}){::}L,\ S\rangle
\end{array}
$$

Fig. 2. The CLS machine.

This abstract machine and its relationship to the operational semantics above provide a convenient starting point from which we can develop an abstract machine for the target language. This machine, defined in section 4, provides a clear description of the storage allocation and deallocation actions provided by stack allocation of variable bindings. Before getting to this implementation we define the static analysis and translation from source language to target language.

3 Static analysis

The goal of our work is to define a translation from source terms to target terms that maximizes the number of annotations. To do this we define a type system in which the types provide information regarding the lifetime of variables. Let Δ (possibly subscripted) range over sets of target language variables.

The set of annotated types is defined as

$$\phi \quad ::= \quad \iota \mid \phi \xrightarrow{\Delta} \phi \mid \phi \xrightarrow{\Delta}_{*} \phi$$

in which ι ranges over some base types. We have two function types to distinguish between functions whose argument does not escape (annotated arrow) and functions whose argument may escape (unannotated arrow). The Δ's over the arrows represent sets of variables. Intuitively, the type $\phi_1 \xrightarrow{\Delta} \phi_2$ represents a function (from type ϕ_1 to ϕ_2) such that Δ contains at least the annotated variables, occurring in the function, which may be accessed during evaluation of the body of the function. We let $\mathsf{LV}(\phi)$ denote the set of *live* annotated term variables occurring in the annotated type ϕ, defined as follows:

$$\mathsf{LV}(\iota) = \{\} \qquad \mathsf{LV}(\phi_1 \xrightarrow{\Delta} \phi_2) = \Delta \cup \mathsf{LV}(\phi_2) \qquad \mathsf{LV}(\phi_1 \xrightarrow{\Delta}_{*} \phi_2) = \Delta \cup \mathsf{LV}(\phi_2)$$

We can relate these annotated types to simple types. Let τ range over simple types including the same set of base types as our annotated types. Then we can define type erasure as a function from annotated types to simple types, written $|\phi|$, as follows

$$|\iota| = \iota \qquad |\phi_1 \xrightarrow{\Delta} \phi_2| = |\phi_1|{\rightarrow}|\phi_2| \qquad |\phi_1 \xrightarrow{\Delta}_* \phi_2| = |\phi_1|{\rightarrow}|\phi_2|$$

We also define erasure on sets of variables. For any set S of target language variables let $|S|$ be the set (of source language variables) obtained by erasing any annotations on variables in S.

We define a translation from source to target language as part of a static analysis using the annotated types. The typing judgment is of the form $\Gamma \rhd e : (\phi, \Delta) \Rightarrow m$ in which

- Γ is a type context, mapping target language variables to types;
- e is a source language expression;
- m is a target language expression;
- ϕ is an annotated type; and
- Δ is a set of annotated language variables.

The judgment can be understood as stating: "Under the type assumptions of Γ, expression e has type ϕ, may access at most the annotated variables in Δ (excluding the bound variables of e) during call-by-value evaluation to weak head normal form (whnf), and translates to target term m." We assume type contexts to be an ordered sequence of bindings with the most recently added binding on the right. We define $\mathsf{dom}(\Gamma) = \{x | (x : \phi) \in \Gamma\} \ \cup \ \{x^* | (x^* {:} \phi) \in \Gamma\}$.

The static analysis is given by the rules in Fig. 3. Before presenting formal properties of this system we provide an informal description of the rules. The first rule (3.1) treats constants. We assume given some signature Σ which maps constants to simple types. A constant c can then be given any annotated type ϕ such that the erasure of ϕ yields the simple type associated with c in Σ. This generality of types of constants is necessary to allow flexible mixing of primitive operators with user-defined functions (which may have annotations and non-empty sets in their types). The next two rules (3.2, 3.3) treat variables. We have two cases to distinguish between the two kinds of variable bindings in contexts and the resulting translation to target language variables. In the rule for the conditional (3.4) observe that we require the types of the two branches to be the same, as expected, but allow the variable sets Δ_2 and Δ_3 to be distinct.

The next two rules (3.5, 3.6) treat λ-abstractions. For both rules, the treatment of the sets Δ and Δ' can easily be explained. We consider just rule (3.5) as it is the one involving an annotated variable. If expression e requires the variables in Δ, then the type of the function is annotated with the set $\Delta' \supseteq \Delta - \{x^*\}$, which includes all the free annotated variables required to evaluate the body of $\lambda x.e$. The expression $\lambda x.e$ requires no variables to be evaluated (because it is already in whnf), so we use $\{\}$ in the conclusion. The ability to weaken the set $\Delta - \{x^*\}$ to Δ' allows certain typing constraints to be satisfied. For example, the two branches of a conditional must have the same type, but they may contain different free variables. If the type of the conditional is functional, then each branch may otherwise (without weakening) have a distinct set of variables annotating the function arrow. The weakening of sets included in rules (3.5) and (3.6) provides a means to construct the same type for each

$$\frac{(c : |\phi|) \in \Sigma}{\Gamma \rhd c : (\phi, \{\}) \Rightarrow c} \tag{3.1}$$

$$\frac{(x^* {:} \phi) \in \Gamma}{\Gamma \rhd x : (\phi, \{x^*\}) \Rightarrow x^*} \qquad \frac{(x {:} \phi) \in \Gamma}{\Gamma \rhd x : (\phi, \{\}) \Rightarrow x} \tag{3.2, 3.3}$$

$$\frac{\begin{array}{c}\Gamma \rhd e_1 : (\mathsf{bool}, \Delta_1) \Rightarrow m_1 \\ \Gamma \rhd e_2 : (\phi, \Delta_2) \Rightarrow m_2 \qquad \Gamma \rhd e_3 : (\phi, \Delta_3) \Rightarrow m_3\end{array}}{\Gamma \rhd \mathsf{if}\ e_1\ e_2\ e_3 : (\phi, \Delta_1 \cup \Delta_2 \cup \Delta_3) \Rightarrow \mathsf{if}\ m_1\ m_2\ m_3} \tag{3.4}$$

$$\frac{\Gamma\{x^* {:} \phi_1\} \rhd e : (\phi_2, \Delta) \Rightarrow m \quad \Delta \subseteq (\Delta' \uplus \{x^*\})}{\Gamma \rhd \lambda x.e : (\phi_1 \xrightarrow{\Delta'}_* \phi_2, \{\}) \Rightarrow \lambda x^*.m}\ x^* \notin \mathsf{LV}(\phi_2) \tag{3.5}$$

$$\frac{\Gamma\{x {:} \phi_1\} \rhd e : (\phi_2, \Delta) \Rightarrow m \quad \Delta \subseteq \Delta'}{\Gamma \rhd \lambda x.e : (\phi_1 \xrightarrow{\Delta'} \phi_2, \{\}) \Rightarrow \lambda x.m} \tag{3.6}$$

$$\frac{\Gamma\{f^* {:} \phi_1 \xrightarrow{\Delta'}_* \phi_2\}\{x^* {:} \phi_1\} \rhd e : (\phi_2, \Delta) \Rightarrow m \quad \Delta \subseteq (\Delta' \uplus \{f^*, x^*\})}{\Gamma \rhd \mu f.\lambda x.e : (\phi_1 \xrightarrow{\Delta'}_* \phi_2, \{\}) \Rightarrow \mu f^*.\lambda x^*.m}\ \begin{array}{l} f^* \notin \mathsf{LV}(\phi_1) \\ x^*, f^* \notin \mathsf{LV}(\phi_2)\end{array} \tag{3.7}$$

$$\frac{\Gamma\{f {:} \phi_1 \xrightarrow{\Delta'} \phi_2\}\{x {:} \phi_1\} \rhd e : (\phi_2, \Delta) \Rightarrow m \quad \Delta \subseteq \Delta'}{\Gamma \rhd \mu f.\lambda x.e : (\phi_1 \xrightarrow{\Delta'} \phi_2, \{\}) \Rightarrow \mu f.\lambda x.m} \tag{3.8}$$

$$\frac{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta}_* \phi_2, \Delta_1) \Rightarrow m_1 \qquad \Gamma \rhd e_2 : (\phi_1, \Delta_2) \Rightarrow m_2}{\Gamma \rhd (e_1 \mathbin{@} e_2) : (\phi_2, \Delta \cup \Delta_1 \cup \Delta_2) \Rightarrow m_1 \mathbin{@^*} m_2} \tag{3.9}$$

$$\frac{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1) \Rightarrow m_1 \qquad \Gamma \rhd e_2 : (\phi_1, \Delta_2) \Rightarrow m_2}{\Gamma \rhd (e_1 \mathbin{@} e_2) : (\phi_2, \Delta \cup \Delta_1 \cup \Delta_2) \Rightarrow m_1 \mathbin{@} m_2} \tag{3.10}$$

Fig. 3. Static analysis and translation.

branch. Weakening is a safe operation because the sets Δ represent a conservative approximation to a set of variables. The approximation is conservative in the sense that if a variable is needed then it will be an element of the set. But not every element of the set is necessarily needed in all possible uses (evaluations) of an expression. This level of imprecision is inherent in the problem due to the expressive power of the language. Our use of weakening follows a similar technique found in Tofte and Talpin (1994) where latent effects over arrows are also allowed to be weakened for similar reasons. Using this technique avoids the need for explicit subtyping as found in Amtoft (1993).

The side condition on rule (3.5), $x^* \notin \mathsf{LV}(\phi_2)$, provides the critical constraint which ensures that the binding for variable x^* can be stack allocated. The key observation here is that the result of evaluating the body e will be a term v of type ϕ_2 (given subject reduction) and if $\mathsf{LV}(\phi_2)$ contains no occurrences of x^* then x^* can never be accessed in the evaluation of any function that may be contained in v. Thus, the binding for x^* can be deallocated (popped) after the value v is produced. Rule

(3.6) contains no side condition and so is always applicable. Note that the names of variables are significant here, and so we should ensure that all bound variable names are distinct. (This assumption is made in section 5 where we prove correctness.)

Rules (3.7, 3.8) treat recursive functions and they operate analogously to the two rules for λ-abstraction. The side conditions on rule (3.7) are precisely those required to ensure that the bindings for both f and x can be allocated on a stack. The condition $f^* \notin \mathsf{LV}(\phi_1)$ ensures that the value (of type ϕ_1) bound to x does not depend upon the value bound to f, as both of these bindings will be on the same stack. Note that the annotation of a recursive function name f does not preclude the binding of another variable to a closure representing a recursive function which can escape. For example consider $(\lambda g.e) @ \mu f^*.\lambda x^*.e'$. The variable g (bound to the closure for the recursive function) may escape the body e. In such a case g could not be annotated. Note that we require the annotations on a recursive function name f to be the same as the (first) argument to the function. While it may be unnecessary, this restriction reduces the number of cases which must be considered. Note further that because we have not included pairs in the language, functions of multiple arguments will be Curried. The current analysis is sensitive to Curried functions: the analysis will never annotate a Curried recursive function of two arguments. Therefore, we should assume that this analysis occurs during compilation only after an unCurry analysis which introduces unCurried forms of functions where possible.

The last two rules (3.9, 3.10) treat application, corresponding to the two forms of application in the target language. In both cases the set $\Delta \cup \Delta_1 \cup \Delta_2$ represents the appropriate set of variables that the application $(e_1 @ e_2)$ may access during evaluation to whnf because the evaluation of $(e_1 @ e_2)$ requires

1. the evaluation of e_2 which requires at most the variables in Δ_2;
2. the evaluation of e_1 which requires at most the variables in Δ_1 to a function $\lambda x.e'$ of type $\phi_1 \xrightarrow{\Delta}_* \phi_2$ or $\phi_1 \xrightarrow{\Delta} \phi_2$; and
3. the evaluation of e' which requires at most the variables in Δ.

The only difference between these two rules is the use of annotations in rule (3.9). If the expression e_1 has type $\phi_1 \xrightarrow{\Delta}_* \phi_2$, then the application in the target language will be annotated.

Some examples of judgments derivable in this system illustrate its ability to detect variables which do not escape their scope and hence can be annotated. For the identity function the following judgment is derivable:

$$\bullet \rhd \lambda x.x : (\phi \xrightarrow{\{\}}_* \phi, \{\}) \quad \Rightarrow \quad \lambda x^*.x^*$$

As should be expected the argument to the identity function can be pushed onto the stack, as no reference to x can remain after evaluating the body of the function. For the function $\lambda x.\lambda y.x$ the following judgment is derivable:

$$\bullet \rhd \lambda x.\lambda y.x : (\phi_1 \xrightarrow{\{\}} (\phi_2 \xrightarrow{\{x\}}_* \phi_1), \{\}) \quad \Rightarrow \quad \lambda x.\lambda y^*.x$$

indicating that only y can be stack allocated. For a slightly more complex example consider the term $\lambda x.((\lambda f.(f @ x)) @ (\lambda z.z))$. Though x has an occurrence inside the body of a locally nested function, once this function is applied to some argument, this occurrence will be accessed. We can derive the judgment

- $\rhd \lambda x.((\lambda f.(f \mathbin{@} x)) \mathbin{@} (\lambda z.z)) : (\phi \xrightarrow{\{\}}_{*} \phi, \{\}) \Rightarrow \lambda x^{*}.((\lambda f^{*}.(f^{*} \mathbin{@}^{*} x^{*})) \mathbin{@}^{*} (\lambda z^{*}.z^{*}))$

indicating that all the variables can be stack allocated. Alternatively, we can derive the judgment

- $\rhd \lambda x.((\lambda f.(f \mathbin{@} (\lambda y.x))) \mathbin{@} (\lambda z.z)) : (\phi_1 \xrightarrow{\{\}} (\phi_2 \xrightarrow{\{x\}}_{*} \phi_1), \{\}) \Rightarrow$
  $\lambda x.((\lambda f^{*}.(f^{*} \mathbin{@}^{*} (\lambda y^{*}.x))) \mathbin{@}^{*} (\lambda z^{*}.z^{*}))$

but not a judgment in which the translated form (for the same input term) contains x annotated. This situation indicates that x cannot be stack allocated. Because the type $(\phi_2 \xrightarrow{\{x\}}_{*} \phi_1)$ contains x, this type would violate the side condition of rule (3.5). Hence, we must use rule (3.6), and the bound variable x cannot be stack allocated. Observe that after applying the source language expression to some value v, we would bind x to v and produce the result $\lambda y.x$. So x cannot be stack allocated.

Before addressing the correctness of our analysis we consider its completeness relative to simply typed terms. We have that every simply typed term can be analyzed and translated into an annotated term. For any context Γ, let $|\Gamma|$ be the context obtained by applying the erasure function to the types in Γ and by erasing any annotations on variables in Γ.

Theorem 3.1 (*Completeness*)

1. If $\vdash \Gamma \rhd e : \tau$ then there exists some Γ', ϕ, Δ, and m such that $\vdash \Gamma' \rhd e : (\phi, \Delta) \Rightarrow m$, $|\Gamma'| = \Gamma$, and $|\phi| = \tau$.
2. if $\vdash \Gamma \rhd e : (\phi, \Delta) \Rightarrow m$ then $\vdash |\Gamma| \rhd e : |\phi|$.

The proof of part (i) relies on constructing function types and variables which are not annotated. Thus, rules (3.2), (3.5) and (3.7) are never used. This restriction also results in all the Δ-sets being empty, leaving essentially the traditional simple-type system. The proof of part (ii) is straightforward as any deduction using the rules of Fig. 3 can be translated into a simple typing derivation by erasing all annotations, sets, and target terms.

While the static analysis has only been specified as a set of inference rules here, this specification does lead to an algorithm for type reconstruction and program translation. The algorithm, which can be found in Burns (1996), is based on algorithm $\mathscr{W}$ (Milner, 1978), but it deals with annotation variables and set variables (and substitutions over these variables) in much the same way in which algorithm $\mathscr{W}$ treats type variables. To handle the problem of efficiently manipulating sets, we employ the technique described in Tofte and Talpin (1994) for managing effects. Our representation of sets takes the form $\delta.\Delta$ in which Δ is a list of variables and δ is a variable. By mapping δ to another representation of a set we can extend the set represented by $\delta.\Delta$ (under some substitution $\mathscr{S}$). The unification of two types may now require two sets $\delta_1.\Delta_1$ and $\delta_2.\Delta_2$ to be unified. This can easily be handled by applying appropriate substitutions to δ_1 and δ_2 (possibly introducing a new set variable δ_3).

4 Implementing the annotated language

The static analysis of the previous section describes a translation from source language to target language. In this section we develop an operational semantics and an abstract machine for the target language. Both use a stack and an environment for allocation of variable bindings. Our goal is not to construct realistic implementations, but to illustrate the use of the analysis and to provide a framework in which we can reason about its correctness. We will avoid discussing specific low-level details about the actual representation of environments and stacks, and, in particular, when such structures can be shared or must be copied.

To construct these specifications, we start with the operational semantics and abstract machine for the source language defined in section 2. Both of these specifications use environments for all binding allocations. We modify these specifications by introducing a stack for allocating the bindings of annotated variables. Unannotated variables will still be allocated in an environment.

4.1 An operational semantics

We introduce the new judgment $\pi;\eta \rhd m \hookrightarrow_t w$ in which π represents a stack, η represents an environment, m is a target term, and w is a target value. We use η instead of ρ to distinguish between an environment which maps variables to source language values (ρ) and one which maps variables to target language values (η). As with environments, we assume that a stack is an ordered sequence of bindings, with the most recent binding on the right end. Thus, pushing the binding $x^* \mapsto w$ onto the stack π yields the stack $\pi\{x^* \mapsto w\}$. We define $\mathsf{dom}(\pi) = \{x^* | (x^* \mapsto w) \in \pi, \text{ for some } w\}$. As in the operational semantics for the source language, values are either constants or function closures of the form $[\eta, \lambda z.m]$ or $[\eta, \mu h \lambda z.m]$. The axiomatization of the judgment $\pi;\eta \rhd m \hookrightarrow_t w$ is given in Fig. 4.

The first rule handles constants. The next two rules (4.2, 4.3) illustrate the two forms of variable access, either from the stack π or the environment η, and the necessity of annotating occurrences of variables in terms to distinguish stack and environment-allocated variables. The next two rules (4.4, 4.5) handle the conditional. The next two rules (4.6, 4.7) illustrate the construction of closures which do not contain the current stack. Our static analysis guarantees that any references in m (the body of the λ-abstraction) to variables on the stack will be evaluated at a point when the variables are still on the stack. Compare this behavior to that of environment-allocated variables, where references to a variable in η cannot necessarily be restricted to a particular segment of a deduction. The next two rules (4.8, 4.9) treat application involving non-recursive function closures. Rule (4.9) is analogous to rule (1.7), using the environment to store the binding. Rule (4.8) illustrates the stack-like behaviour of annotated variables. The binding $x \mapsto w_2$ is pushed onto the stack just prior to evaluating m'. Upon computing the value w, we can pop the stack, returning it to its form just prior to evaluating $(m_1 @^* m_2)$. The action of popping the stack is implicit in this specification.

The final two rules (4.10, 4.11) treat application involving recursive function

$$\frac{}{\pi ; \eta \ \triangleright c \hookrightarrow_t c} \tag{4.1}$$

$$\frac{\pi(x^*) = w}{\pi ; \eta \ \triangleright x^* \hookrightarrow_t w} \qquad \frac{\eta(x) = w}{\pi ; \eta \ \triangleright x \hookrightarrow_t w} \tag{4.2, 4.3}$$

$$\frac{\pi ; \eta \ \triangleright m_1 \hookrightarrow_t \mathsf{true} \qquad \pi ; \eta \ \triangleright m_2 \hookrightarrow_t w}{\pi ; \eta \ \triangleright \mathsf{if}\ m_1\ m_2\ m_3 \hookrightarrow_t w} \tag{4.4}$$

$$\frac{\pi ; \eta \ \triangleright m_1 \hookrightarrow_t \mathsf{false} \qquad \pi ; \eta \ \triangleright m_3 \hookrightarrow_t w}{\pi ; \eta \ \triangleright \mathsf{if}\ m_1\ m_2\ m_3 \hookrightarrow_t w} \tag{4.5}$$

$$\frac{}{\pi ; \eta \ \triangleright \lambda z.m \hookrightarrow_t [\eta, \lambda z.m]} \qquad \frac{}{\pi ; \eta \ \triangleright \mu h.\lambda z.m \hookrightarrow_t [\eta, \mu h.\lambda z.m]} \tag{4.6, 4.7}$$

$$\frac{\begin{array}{c}\pi ; \eta \ \triangleright m_1 \hookrightarrow_t [\eta', \lambda x^*.m'] \qquad \pi ; \eta \ \triangleright m_2 \hookrightarrow_t w_2 \\ \pi\{x^* {\mapsto} w_2\};\ \eta' \ \triangleright m' \hookrightarrow_t w\end{array}}{\pi ; \eta \ \triangleright m_1 \ @^*\ m_2 \hookrightarrow_t w} \tag{4.8}$$

$$\frac{\begin{array}{c}\pi ; \eta \ \triangleright m_1 \hookrightarrow_t [\eta', \lambda x.m'] \qquad \pi ; \eta \ \triangleright m_2 \hookrightarrow_t w_2 \\ \pi;\ \eta'\{x {\mapsto} w_2\} \ \triangleright m' \hookrightarrow_t w\end{array}}{\pi ; \eta \ \triangleright m_1 \ @\ m_2 \hookrightarrow_t w} \tag{4.9}$$

$$\frac{\begin{array}{c}\pi ; \eta \ \triangleright m_1 \hookrightarrow_t [\eta', \mu f^*.\lambda x^*.m'] \qquad \pi ; \eta \ \triangleright m_2 \hookrightarrow_t w_2 \\ \pi\{f^* {\mapsto} [\eta', \lambda x^*.m']\}\{x^* {\mapsto} w_2\};\ \eta' \ \triangleright m' \hookrightarrow_t w\end{array}}{\pi ; \eta \ \triangleright m_1 \ @^*\ m_2 \hookrightarrow_t w} \tag{4.10}$$

$$\frac{\begin{array}{c}\pi ; \eta \ \triangleright m_1 \hookrightarrow_t [\eta', \mu f.\lambda x.m'] \qquad \pi ; \eta \ \triangleright m_2 \hookrightarrow_t w_2 \\ \pi;\ \eta'\{f \mapsto [\eta', \mu f.\lambda x.m']\}\{x {\mapsto} w_2\} \ \triangleright m' \hookrightarrow_t w\end{array}}{\pi ; \eta \ \triangleright m_1 \ @\ m_2 \hookrightarrow_t w} \tag{4.11}$$

Fig. 4. Target language operational semantics.

closures. These rules are analogous to the previous two, except for the addition of handling the recursive function binding. Note that the binding for f^* pushed on the stack binds f^* to the closure $\left[\eta', \lambda x^*.m'\right]$, not $\left[\eta', \mu f^*.\lambda x^*.m'\right]$. This is possible because recursive calls to f^* in m' need not push another binding for f^* onto the stack each time. Thus only upon the first invocation of a recursive function is the closure $\left[\eta', \mu f^*.\lambda x^*.m'\right]$ encountered and a binding for f^* pushed.

4.2 An abstract machine

While the operational semantics for the target language provides some of the flavor of the distinction between stack and environment allocation, an abstract machine description can further illustrate the stack behavior. We can adapt the CLS machine from section 2 to manipulate both a stack and a environment. The new machine, the CπLS machine, uses a state of the form $\langle C, \pi, L, S\rangle$ in which we have added the variable stack π. We introduce the new instructions aps and pop. The aps instruction pushes a binding onto the stack, and the pop instruction pops the stack. The CπLS machine is given in Fig. 5.

$$
\begin{array}{lcl}
\langle c{::}C,\ \pi,\ \eta{::}L,\ S\rangle & \Rightarrow & \langle C,\ \pi,\ L,\ c{::}S\rangle \\
\langle x^{*}{::}C,\ \pi,\ \eta{::}L,\ S\rangle & \Rightarrow & \langle C,\ \pi,\ L,\ \pi(x^{*}){::}S\rangle \\
\langle x{::}C,\ \pi,\ \eta{::}L,\ S\rangle & \Rightarrow & \langle C,\ \pi,\ L,\ \eta(x){::}S\rangle \\
\langle(\mathsf{if}\ e_1\ e_2\ e_3){::}C,\pi,\eta{::}L,\ S\rangle & \Rightarrow & \langle e_1{::}\mathsf{branch}(e_2,e_3){::}C,\ \pi,\eta{::}\eta{::}L,\ S\rangle \\
\langle\mathsf{branch}(e_2,e_3){::}C,\pi,\eta{::}L,\ \mathsf{true}{::}S\rangle & \Rightarrow & \langle e_2{::}C,\ \pi,\eta{::}L,\ S\rangle \\
\langle\mathsf{branch}(e_2,e_3){::}C,\ \pi,\eta{::}L,\ \mathsf{false}{::}S\rangle & \Rightarrow & \langle e_3{::}C,\ \pi,\ \eta{::}L,\ S\rangle \\
\langle(\lambda z.m){::}C,\ \pi,\ \eta{::}L,\ S\rangle & \Rightarrow & \langle C,\ \pi,\ L,\ [\eta,\lambda z.m]\ {::}S\rangle \\
\langle(\mu f.\lambda x.m){::}C,\ \pi,\ \eta{::}L,\ S\rangle & \Rightarrow & \langle C,\ \pi,\ L,\ [\eta,\mu f.\lambda x.m]\ {::}S\rangle \\
\langle(\mu f^{*}.\lambda x^{*}.m){::}C,\ \pi,\ \eta{::}L,\ S\rangle & \Rightarrow & \langle C,\ \pi,\ L,\ [\eta,\mu f^{*}.\lambda x^{*}.m]\ {::}S\rangle \\
\langle(m_1\ @^{*}\ m_2){::}C,\ \pi,\ \eta{::}L,\ S\rangle & \Rightarrow & \langle m_2{::}m_1{::}\mathsf{aps}{::}C,\pi,\ \ \eta{::}\eta{::}L,\ S\rangle \\
\langle(m_1\ @\ m_2){::}C,\ \pi,\ \eta{::}L,\ S\rangle & \Rightarrow & \langle m_2{::}m_1{::}\mathsf{ap}{::}C,\pi,\ \ \eta{::}\eta{::}L,\ S\rangle \\
\langle\mathsf{aps}{::}C,\ \pi,\ L,\ [\eta,\lambda x^{*}.m]\ {::}w{::}S\rangle & \Rightarrow & \langle m{::}\mathsf{pop}{::}C,\ \pi\{x^{*}\mapsto w\},\ \eta{::}L,\ S\rangle \\
\langle\mathsf{ap}{::}C,\ \pi,\ L,\ [\eta,\lambda x.m]\ {::}w{::}S\rangle & \Rightarrow & \langle m{::}C,\ \pi,\ (\eta\{x\mapsto w\}){::}L,\ S\rangle \\
\langle\mathsf{aps}{::}C,\ \pi,\ L,\ [\eta,\mu f^{*}.\lambda x^{*}.m]\ {::}w{::}S\rangle & \Rightarrow & \langle m{::}\mathsf{pop}{::}\mathsf{pop}{::}C,\ \pi\{f^{*}\mapsto[\eta,\lambda x^{*}.m]\}\{x^{*}\mapsto w\},\ \eta{::}L,\ S\rangle \\
\langle\mathsf{ap}{::}C,\ \pi,\ L,\ [\eta,\mu f.\lambda x.m]\ {::}w{::}S\rangle & \Rightarrow & \langle m{::}C,\ \pi,\ (\eta\{f\mapsto[\eta,\mu f.\lambda x.m]\}\{x\mapsto w\}){::}L,S\rangle \\
\langle\mathsf{pop}{::}C,\ \pi\{x\mapsto w\},\ L,\ S\rangle & \Rightarrow & \langle C,\ \pi,\ L,\ S\rangle
\end{array}
$$

Fig. 5. The CπLS machine.

The CπLS machine has two rules for handling variables, two rules for applications and two rules for function application. In each case, one of the two rules manipulates unannotated terms just as in the CLS machine, while the other rule manipulates annotated terms. Of particular interest are the two rules involving application. The rule for reducing an annotated application introduces the aps instruction when it decomposes the term $(m_1\ @^{*}\ m_2)$. This indicates that the value of m_2 will be pushed onto the stack during evaluation and should be popped after the value of the application is produced. The first two rules for aps and ap illustrate the difference in storage allocation, with the first one pushing the binding onto the stack π, and the second one adding the binding to the environment η. The next two rules illustrate the handling of recursive function closures by pushing a binding for the recursive function onto the stack. The addition of pop instructions ensures that the two bindings will be popped when the recursive function terminates.

We have implemented this machine using a syntax with de Bruijn indices in place of symbolic variable names. This eliminates the need to search for variable names, but it also complicates the handling of stacks and environments because the size of the stack may grow between the time a function (which contains de Bruijn indices which index into the stack) is first encountered and when it is actually applied.

We can prove the correctness of this abstract machine with respect to the operational semantics of Fig. 4, using an approach analogous to the one we took in Hannan and Pfenning (1992), in which we proved the correctness of the CLS machine with respect to the operational semantics of Fig. 1. The difficulty in relating the abstract machine to its operational semantics arises from two differences: (1) the operational semantics axiomatizes a judgment relating an expression and its final value while the abstract machine axiomatizes a relation between states in a machine in which one state is obtained from the other by a single reduction step; and (2) the abstract machine state may contain contextual information relating to previous (S) and subsequent (C) computations while the operational semantics does not. We surmount these difficulties by isolating an invariant of the abstract machine, namely that starting from a machine state $\langle m::C,\ \pi,\ \eta::L,\ S\rangle$, the machine will eventually reach a state $\langle C,\ \pi,\ L,\ w::S\rangle$ where w is the value of m (assuming m has a value). This property motivates the following theorem.

Theorem 4.1
$\vdash \pi;\eta \rhd m \hookrightarrow_t w$ iff for all C, L, S, $\langle m::C,\ \pi,\ \eta::L,\ S\rangle \Rightarrow^* \langle C,\ \pi,\ L,\ w::S\rangle$.

The proof is a straightforward extension of our previous proof of correctness for the CLS machine (Hannan and Pfenning, 1992), where some insight is needed to capture the nature of the stack discipline of the abstract machine. In the forward direction the proof uses induction on the structure of the deduction for $\pi;\eta \rhd m \hookrightarrow_t w$. In the reverse direction the proof uses induction on the length of the reduction $\langle m::C,\ \pi,\ \eta::L,\ S\rangle \Rightarrow^* \langle C,\ \pi,\ L,\ w::S\rangle$. Observe that a consequence of this theorem is that the abstract machine never reaches a stuck state such as $\left\langle \mathsf{aps}::C,\ \pi,\ \eta::L,\ [\eta', \lambda x.m]::w::S\right\rangle$: the operator term of an annotated application cannot reduce to an unannotated function closure.

Of particular interest is the case in which the C, L, and S are all empty. From this we observe that to evaluate expression m with respect to stack π and environment η we should start the machine in the state $\langle m::\mathsf{nil},\ \pi,\ \eta::\mathsf{nil},\ \mathsf{nil}\rangle$ and if m has a value w, the machine will stop in the state $\langle \mathsf{nil},\ \pi,\ \mathsf{nil},\ w::\mathsf{nil}\rangle$.

4.3 Two simple optimizations

The annotated language contains two forms of application, distinguishing between applications in which the operator part evaluates to a stackable function and applications in which the operator part evaluates to a regular function. The annotated form of application, however, is not necessary. We can redefine our analysis and target language implementations without using $@^*$, while still maintaining all of our results. While this modification has little impact on the limited language considered here, it can be significant on a larger language.

We begin by modifying rule (3.7) of Fig. 3 simply by replacing $@^*$ with $@$:

$$\frac{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta}_* \phi_2, \Delta_1) \Rightarrow m_1 \qquad \Gamma \rhd e_2 : (\phi_1, \Delta_2) \Rightarrow m_2}{\Gamma \rhd (e_1 \mathbin{@} e_2) : (\phi_2, \Delta \cup \Delta_1 \cup \Delta_2) \Rightarrow m_1 \mathbin{@} m_2}$$

Next we modify the operational semantics of Fig. 4 by replacing rule (4.8) with

$$\frac{\pi;\eta \rhd m_1 \hookrightarrow_t [\eta', \lambda x^*.m'] \qquad \pi;\eta \rhd m_2 \hookrightarrow_t w_2 \qquad \pi\{x^* \mapsto w_2\};\ \eta' \rhd m' \hookrightarrow_t w}{\pi;\eta \rhd m_1 \mathbin{@} m_2 \hookrightarrow_t w}$$

Again, we have simply replaced $@^*$ with $@$. In the resulting operational semantics we now have two rules for evaluating an application $(m_1 \mathbin{@} m_2)$. Deciding which rule to apply is based on whether m_1 evaluates to an annotated or unannotated

λ-abstraction. In both rules, we evaluate both m_1 and m_2 in the same ways. Once we have values for these terms we can decide which rule to use. Using these observations we can redesign the relevant rules of Fig. 5 for the CπLS machine. The current rules for reducing the two forms of application differ in the use of the @ and @* instructions. As expected, the decision as to whether the binding for value of m_2 will be placed on the stack or environment is fixed at this point. We can delay this decision until after we have the value of m_1. The natural point to make this decision is during the reduction of the ap instruction. Using these observations we replace four rules for applications, aps, and ap with the following new rules:

$$\begin{array}{lcl}
\langle (m_1 \,@\, m_2)::C,\ \pi,\ \eta::L,\ S\rangle & \Rightarrow & \langle m_2::m_1::\mathsf{ap}::C,\ \pi,\ \eta::\eta::L,\ S\rangle \\
\left\langle \mathsf{ap}::C,\ \pi,\ L,\ \left[\eta,\lambda x^*.m\right]::w::S\right\rangle & \Rightarrow & \langle m::\mathsf{pop}::C,\ \pi\{x^*\mapsto w\},\ \eta::L,\ S\rangle \\
\langle \mathsf{ap}::C,\ \pi,\ L,\ [\eta,\lambda x.m]::w::S\rangle & \Rightarrow & \langle m::C,\ \pi,\ (\eta\{x\mapsto w\})::L,\ S\rangle
\end{array}$$

Application is reduced in only one way, and now the ap instruction, based on the structure of the value of m_1 which is on the stack, decides whether to insert a pop instruction. This optimization of the machine can be exploited as follows: a term could be of the form ((if m_1 then m_2 else m_3) @ m_4) in which m_2 evaluates to a function (closure) $\lambda x^*.w_2$ while m_3 evaluates to a function (closure) $\lambda x.w_3$. Then the value of m_4 could be pushed onto the stack in the case that m_1 evaluates to true, while it could be placed in the environment in the case that m_1 evaluates to false. Using the unoptimized machine we would be forced to remove the annotations from $\lambda x^*.w_2$, even though x^* is stackable. Similar results hold for recursive functions.

If we extend the target operational semantics to support a tail recursion optimization then we can extend the analysis to identify tail-recursive function calls. If an application is identified as being a tail recursive call (and the argument to the function has been allocated on the stack) then the standard optimization can be performed. Assume '@^{t}' is used to denote tail calls. (The identification of such calls is straightforward and not presented here.) We introduce the new instruction apt and add the following rules to the abstract machine of Fig. 5:

$$\begin{array}{lcl}
\langle (m_1 \,@^t\, m_2)::C,\ \pi,\ \eta::L,\ S\rangle & \Rightarrow & \langle m_2::m_1::\mathsf{apt}::C,\ \pi,\ \eta::\eta::L,\ S\rangle \\
\left\langle \mathsf{apt}::C,\ \pi\{x^*\mapsto w_x\},\ L,\ \left[\eta,\lambda x^*.m\right]::w::S\right\rangle & \Rightarrow & \langle m::C,\ \pi\{x^*\mapsto w\},\ \eta::L,\ S\rangle
\end{array}$$

Note that no pop instruction is generated in this case.

5 Correctness

To justify our static analysis and show the correctness of the target language operational semantics (with respect to the source language semantics) we need to demonstrate a consistency among the static analysis and the two operational semantics. In particular we must show that the annotations on types have an interpretation consistent with the informal description given to them earlier. For example, if a term e can be shown to have type $\phi_1 \xrightarrow{\Delta}_* \phi_2$ by our static analysis and if e evaluates to value v, then we want to ensure that v is a function closure consistent with the type $\phi_1 \xrightarrow{\Delta}_* \phi_2$, taking into account the annotation Δ. Our approach to showing correctness is based on the proof of consistency between a static and operational semantics from Milner and Tofte (1991), where a polymorphic language is shown to be consistent with its type system. We adapt this proof by simultaneously considering two operational semantics. Our correctness result will also imply the consistency (in the sense of Milner and Tofte (1991)) of both the source and target languages with respect to the static analysis.

We make one assumption about the names of variables in terms we manipulate. We assume that each λ (and μ) binds a variable which is distinct from all the other bound variables and distinct from all the free variables. This is not a significant restriction as we can always α-convert terms to achieve this condition. Let $\mathsf{Rectify}(e)$ be true of a term e when all the bound variables in e are unique and distinct from all the free variables of e. Any term not in such a form can be α-converted to satisfy Rectify. We assume that a term has been rectified before undergoing the static analysis. If we start with a rectified term then we will be able to assume that bindings in environments are distinct, as described by the following. We not only require terms to be rectified, but we also require a similar property of environments, contexts, and values.

Definition 5.1 (*Distinct*)
The property Distinct is the smallest property satisfying

1. $\mathsf{Distinct}(\bullet)$ for empty environments and empty contexts;
2. $\mathsf{Distinct}(\rho\{x \mapsto v\})$ if $\mathsf{Distinct}(\rho)$, $\mathsf{Distinct}(v)$, and $x \notin \mathsf{dom}(\rho)$;
3. $\mathsf{Distinct}(\Gamma\{x : \phi\})$ if $\mathsf{Distinct}(\Gamma)$ and $x \notin |\mathsf{dom}(\Gamma)|$;
4. $\mathsf{Distinct}(\Gamma\{x^* : \phi\})$ if $\mathsf{Distinct}(\Gamma)$ and $x \notin |\mathsf{dom}(\Gamma)|$;
5. $\mathsf{Distinct}(c)$ for all constants c;
6. $\mathsf{Distinct}([\rho, e])$ if $\mathsf{Distinct}(\rho)$, $\mathsf{Rectify}(e)$, and $\mathsf{BV}(e) \cap \mathsf{dom}(\rho) = \{\}$.

The last case in this definition describes the required property of function closures: the environment must be distinct, the term must be rectified, and the bound variables occurring in the term must be distinct from the variables bound in the environment. This last condition ensures that if we extend the environment with a (currently) bound variable from the term, then the resulting environment will also be distinct (assuming the value bound to the variable is also distinct).

A useful property of the operational semantics for the source language is that the distinct property is preserved by evaluation.

Theorem 5.2 (*Distinct Source Environments*)
If $\mathsf{Distinct}([\rho, e])$ and $\Pi :: \rho \rhd e \hookrightarrow v$ then $\mathsf{Distinct}(v)$.

The proof is straightforward by induction on Π.

We next introduce a definition for the consistency of values in the source and target languages. A traditional, type-based consistency relation found in proofs of type soundness (subject reduction) uses a binary relation between values and types, e.g. $v : \tau$. When we add a source and target language we naturally have two values (one in the source language and one in the target language) related to a type, e.g. $v \approx w : \tau$. When values include closures, the consistency relation is extended to environments and contexts, e.g. $\rho \approx \eta : \Gamma$. However, several aspects of our analysis and target language complicate this simple notion of consistency. First, values in our target language include closures $[\eta, m]$ in which m can contain annotated variables whose values are found on a stack (and not in η). Hence consistency between values must be relative to a stack. Second, the relationship between variable bindings in a source environment and a target environment and stack is not a trivial one-to-one relationship. The ability to pop a stack can allow fewer bindings in the target setting than in the source setting. The 'global' nature of the stack in the target language can also result in bindings in π with no corresponding bind in ρ. (This happens during some computation using a local environment ρ.) To account for these discrepancies we define consistency relative to a set of variables Δ. The set Δ includes those (annotated) variables on the stack whose binding is required to establish the consistency of two values. If a variable x is not in Δ then the binding of x will not be needed (to establish consistency). The resulting consistency relation for values is a quintuple $\mathsf{Consistent}(\pi, v, w, \phi, \Delta)$. The relation for environments, stacks and contexts is also a quintuple $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$.

Definition 5.3 (*Consistency*)
The relations $\mathsf{Consistent}(\pi, v, w, \phi, \Delta)$ and $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ are the two smallest relations satisfying

1. $\mathsf{Consistent}(\pi, c, c, \phi, \Delta)$ for all π, Δ and $(c : |\phi|) \in \Sigma$;
2. $\mathsf{Consistent}(\pi, [\rho, \lambda x.e], [\eta, \lambda z.m], \phi, \Delta)$ if there exists a Γ such that $\Gamma \rhd \lambda x.e : (\phi, \{\}) \Rightarrow \lambda z.m$ is derivable, and $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta \cap \mathsf{LV}(\phi))$;
3. (a) $\mathsf{Consistent}(\pi, [\rho, \mu f.\lambda x.e], [\eta, \mu h.\lambda z.m], \phi, \Delta)$ if there exists a Γ such that $\Gamma \rhd \mu f.\lambda x.e : (\phi, \{\}) \Rightarrow \mu h.\lambda z.m$ is derivable, and $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta \cap \mathsf{LV}(\phi))$;
   (b) $\mathsf{Consistent}(\pi, [\rho, \mu f.\lambda x.e], [\eta, \lambda x^*.m], \phi, \Delta)$ if there exists a Γ such that $\Gamma \rhd \mu f.\lambda x.e : (\phi, \{\}) \Rightarrow \mu f^*.\lambda x^*.m$ is derivable, and $\mathsf{Consistent}(\rho\{f \mapsto [\rho, \mu f.\lambda x.e']\}, \pi, \eta, \Gamma\{f^* : \phi\}, (\Delta \cap \mathsf{LV}(\phi)) \uplus \{f^*\})$;
4. $\mathsf{Consistent}(\bullet, \bullet, \bullet, \bullet, \{\})$;
5. $\mathsf{Consistent}(\rho\{x \mapsto v\}, \pi, \eta\{x \mapsto w\}, \Gamma\{x : \phi\}, \Delta)$ if $(\mathsf{LV}(\phi) \cap \Delta) \subseteq \mathsf{dom}(\Gamma)$, $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ and $\mathsf{Consistent}(\pi, v, w, \phi, \Delta)$;
6. $\mathsf{Consistent}(\rho\{x \mapsto v\}, \pi\{x^* \mapsto w\}, \eta, \Gamma\{x^* : \phi\}, \Delta \uplus \{x^*\})$ if $(\mathsf{LV}(\phi) \cap \Delta) \subseteq \mathsf{dom}(\Gamma)$, $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ and $\mathsf{Consistent}(\pi, v, w, \phi, \Delta)$;
7. $\mathsf{Consistent}(\rho\{f \mapsto [\rho', \mu f.\lambda x.e]\}, \pi\{f^* \mapsto [\eta', \lambda x^*.m]\}, \eta, \Gamma\{f^* : \phi\}, \Delta \uplus \{f^*\})$ if $(\mathsf{LV}(\phi) \cap \Delta) \subseteq \mathsf{dom}(\Gamma)$, $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ and $\mathsf{Consistent}(\pi, [\rho', \mu f.\lambda x.e], [\eta', \mu f^*.\lambda x^*.m], \phi, \Delta)$;
8. $\mathsf{Consistent}(\rho\{x \mapsto v\}, \pi, \eta, \Gamma\{x^* : \phi\}, \Delta)$ if $(\mathsf{LV}(\phi) \cap \Delta) \subseteq \mathsf{dom}(\Gamma)$ and $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$;
9. $\mathsf{Consistent}(\rho, \pi\{x^* \mapsto w\}, \eta, \Gamma, \Delta)$ if $x^* \notin \Delta$, and $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$.

Note that because environments (and all the other data structures involved) are well-founded this definition is also well-founded, and hence we can reason by induction over derivations of these judgments.

This definition is somewhat involved and deserves explanation, as it is critical to the correctness proof. The set Δ represents the annotated variables which may be used in the future, and so it is only these annotated variables whose values must be shown to be consistent. The first three rules describe the consistency of two values. Constants can only be related to themselves, independent of π and Δ. The consistency of two closures (rules (2) and (3)) requires a typing derivation and the consistency of environments, stacks, and contexts, with respect to $\Delta \cap \mathsf{LV}(\phi)$. This restriction of Δ to just the live variables of ϕ allows us to eliminate annotated variables which might have been popped from the stack. Note that rule (3b) relates a recursive function closure from the source language with a non-recursive function closure from the target language. This relationship is required to account for the optimization of the target language which replaces a recursive function closure with a non-recursive one (rule (4.10)).

The rules of consistency for environments, stacks and contexts can be motivated as follows. Rules (5) and (6) provide the means for extending environments, stacks and contexts (with arguments to functions) during function application. The condition $(\mathsf{LV}(\phi) \cap \Delta) \subseteq \mathsf{dom}(\Gamma)$ ensures that the annotated variables of interest in w all occur in Γ. Rule (7) relates a recursive function closure in the source environment with a non-recursive target-language function closure in π. We require this rule due to the different handling of recursive function closures in the two operational semantics (rules (1.8) and (4.10)). Rule (8) allows for consistency to hold after an item has been popped from the stack (but still exists in ρ and Γ). Rule (9) allows for consistency when the stack contains global bindings which have no corresponding bindings in a local environment ρ and local context Γ. The condition $x^* \notin \Delta$ implies that the value bound to x^* is not needed to show consistency.

While this definition of consistency may appear overly complicated, it precisely captures the relationship between environments, stacks, and contexts. Note that $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ implies $\mathsf{dom}(\rho) = \mathsf{dom}(\Gamma)$ and $\Delta \subseteq \mathsf{dom}(\Gamma)$. We establish these and other properties in the appendix, where they contribute to the proof of the theorem below.

We can now give the main correctness theorem. The theorem requires a number of technical conditions on the relationships between environments, stacks, sets, etc. Note that these conditions are trivially satisfied when we start with a closed, rectified term and empty environments and stacks.

Theorem 5.4

If $\mathsf{Distinct}([\rho, e])$, $\Xi :: \Gamma \rhd e : (\phi, \Delta_0) \Rightarrow m$, $\Delta_0 \subseteq \Delta'$, and $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta')$ then

1. $\Pi :: \rho \rhd e \hookrightarrow v$ implies that there exist w and $\Theta :: \pi ; \eta \rhd m \hookrightarrow_t w$ such that $\mathsf{Consistent}(\pi, v, w, \phi, \Delta')$;
2. $\Theta :: \pi ; \eta \rhd m \hookrightarrow_t w$ implies that there exist v and $\Pi :: \rho \rhd e \hookrightarrow v$ such that $\mathsf{Consistent}(\pi, v, w, \phi, \Delta')$.

The proof can be found in the appendix.

From this general theorem (and the correctness of the CLS machine of Fig. 2) we can state a simpler statement of correctness. Given a closed λ-term of a base type, we have the following corollary.

Corollary 5.5
If $\mathsf{Rectify}(e)$ and $\vdash \bullet \ \rhd e : (\iota, \Delta) \Rightarrow m$ for some base type ι then

1. $\vdash \bullet \ \rhd e \hookrightarrow c$ iff $\vdash \bullet ; \bullet \ \rhd m \hookrightarrow_t c$; and
2. $\langle e::\mathsf{nil}, \rho::\mathsf{nil}, \mathsf{nil}\rangle \Rightarrow \langle \mathsf{nil}, \mathsf{nil}, v::\mathsf{nil}\rangle$ iff $\langle m::\mathsf{nil}, \pi, \eta::\mathsf{nil}, \mathsf{nil}\rangle \Rightarrow \langle \mathsf{nil}, \pi, \mathsf{nil}, w::\mathsf{nil}\rangle$

for some constant $(c : \iota) \in \Sigma$.

Because only constants can have a base types and the consistency of base-type values is essentially the identity relation, we are guaranteed to have both source and target languages produce the same constant.

The proof of Theorem 5.4 and all the supporting lemmas and propositions have been encoded into the programming language Elf and partially machine-checked (Ibarra, 1997), using techniques developed in previous work (Hannan and Pfenning, 1992). In this setting, we encode proofs as sets of type declarations. The types correspond to formulas (representing theorems, lemmas, etc.) and the objects of these types correspond to cases of the proofs. The implementation of the Elf language provides type reconstruction and type checking, which serves to guarantee the correctness of each case. However, the system does not guarantee that all cases of a proof have been included. Hence, machine-checking alone is not sufficient. Visual inspection of the code ensures that all cases have been considered.

The encoding of the proofs proceeded by first translating the syntaxes, semantics, and various definitions, and then sequentially translating the propositions, lemma, main theorem, and their corresponding proofs. In addition, we needed to encode numerous propositions regarding the manipulation of sets (e.g. the associative property of set intersection). While generally not difficult, this task proved tedious and time-consuming. During the encoding process we uncovered a few typographical errors in the paper and several minor errors in the original definitions of the analysis and of consistency. The total effort of encoding the proofs as described in Ibarra (1997) required approximately two months by an individual who had no prior experience with Elf or theorem-provers. Subsequent revisions based on modifications of the analysis and proofs required an additional week. While often tedious work, the process of encoding these proofs served to deepen our understanding of the requirements of the analysis and definition of consistency. The explicit manipulation of Elf objects representing deductions in these systems yielded an appreciation for every single aspect of these definitions, serving to strengthen our belief in the correctness of these definitions.

6 Future work

The current analysis provides an escape analysis in which each bound variable is determined not to escape or possibly to escape its scope. This provides useful information for handling non-escaping variables (such as stack allocation), but provides no information to support the efficient handling of escaping variables. A useful refinement of the analysis is to provide lifetime information for these escaping variables. Such information could be used to determine closure representations, and, in some cases, to determine that a form of stack allocation is still possible.

Another extension to the analysis is the support of polymorphism. Traditional type polymorphism does not appear to present any significant challenges to the escape analysis. The instantiation of a type variable with an annotated type and the quantification of a type variable can be carried out in the usual fashion. The ability to instantiate a type variable with either an annotated or unannotated function type simply means that some decisions about allocation strategies must be performed at run time. With annotated types we can consider the introduction of set variables and their quantification. This kind of set polymorphism allows us to generalize over the sets annotating function types. The ability to do this can potentially provide better information about variable usage. In particular, quantified set variables contribute to the support of separate compilation.

Consider the example of function composition. Supporting set variables and their quantification, our analysis should specify:

$$\texttt{compose} : \forall\alpha\forall\beta\forall\gamma\forall\delta_1\forall\delta_2.(\alpha \xrightarrow{\delta_1} \beta) \xrightarrow{\{\}} (\beta \xrightarrow{\delta_2} \gamma) \xrightarrow{\{\}} (\alpha \xrightarrow{\delta_1\cup\delta_2} \gamma)$$

Without such quantification of set variables, the type of `compose` would depend on all of its possible uses. Each use may contribute variables to the sets annotating the function type. By introducing quantified set variables we can avoid this dependency and give the function a set-polymorphic type independent of its uses. Extending the traditional polymorphic type schemas of Standard ML with quantification over set variables and extending the static analysis to include generalization and instantation rules (similar to those used in Standard ML's type system), we can produce types such as the one above for `compose`.

7 Related work

This work considers the same problem addressed in Banerjee and Schmidt (1994), though in that work the authors base their analysis on Sestoft's closure analysis (Sestoft, 1991). This closure analysis computes the fixed points of two equations that describe the set of closures to which an expression can evaluate and the set of closures to which a variable can be bound. This analysis appears to be more precise than our analysis as we only generate a set of variables, along with a type, with no indication of precisely where these variables occurred. For the current application of stack allocation, however, this is all that is required. Their analysis appears to correspond to our analysis producing a target term that is completely annotated (meaning that all allocations can be done on a stack), though the precise correspondence between the two approaches has not been considered. However, closure analysis is typically a global operation and hence does not support separate compilation. As mentioned in the previous section, our type-based analysis can support separate compilation.

Most closely related to our work is the escape analysis of Goldberg and Park

(1990), which starts with a denotational semantics and develops an escape analysis based on abstract interpretation techniques. Their approach supports an analysis similar in spirit to ours in that it examines the properties of a function independent of how it is used. Their approach also supports a local analysis which examines particular applications of functions. This local analysis can provide better results than our approach. For example, in the expression $(((\lambda x.\lambda y.x + y)\, e_1)\, e_2)$ our analysis would indicate that x can escape its scope. The local analysis of Goldberg and Park (1990) detects that stack allocation of x is still possible. However, their approach again appears to require global information, and, thus, does not support separate compilation.

Recent work on region inference (Tofte and Talpin, 1994) provides a more sophisticated storage allocation analysis. In this work a type system guides the translation from a functional language to a target language which contains explicit references to blocks or regions *in which values are stored*. The storage model used to implement this target language uses a stack of regions, where regions can be of varying size. The translation to the target language incorporates a lifetime analysis, detecting expressions in which any storage allocated can be deallocated upon completing the evaluation of the expression. In our work we have so far ignored the issue of storage for values, choosing to focus only on the allocation of bindings. As mentioned in the introduction our analysis differs from region analysis in that we study the lifetime of variable bindings, not values. Besides this distinction, the present analysis only differentiates between escaping and non-escaping bindings, while region analysis studies the lifetime of values and partitions values into regions based on their similar lifetimes. These two efforts, however, are complementary as both can be used together to support efficient implementations of functional languages. Region analysis supports efficient allocation of the store while our escape analysis supports efficient allocation (representation) of the environment.

The use of sets of relevant variables is also present in the type reconstruction system of Damas (1985), where the sets are used to record information about reference variables that might be updated during evaluation. The use of two kinds of arrow types is present in Amtoft (1993), where the task considered there is the process of converting call-by-name into call-by-value. Though a different problem domain than ours, the two have much in common. Both have a ‘default’ approximation – environment allocation here and call-by-name there – and both try to find better approximations by finding those instances of terms which have certain properties. Like the type system used here, Amtoft uses annotated types to distinguish between two cases (call-by-value and call-by-name). He mentions the possible use of annotating arrows with sets of variables, but dismisses it because of complications. More generally, our use of type systems to perform a static analysis joins a growing list of such efforts, including Baker-Finch (1994), Kuo and Mishra (1989) and Wright (1991), as well as those mentioned above.

As mentioned in the introduction, this work has impact on closure conversion because stack allocated variables need not be included in closures. From this perspective, our work relates to other work on closure conversion, particularly approaches to detecting variables which need not be included in closures (Hannan, 1995; Wand and Steckler, 1997).

8 Conclusion

We have presented a type system that provides an analysis of a simple functional language and yields a translation from this language to an annotated version of the language. The type system guarantees that the annotations can be given interpretations that allow the stack-allocation of variable bindings. Starting with an operational semantics and an abstract machine for the source language, we developed corresponding operational semantics and abstract machine for the annotated language, introducing a stack for variable allocations, but also retaining the environment. These implementations of the annotated language serve primarily as a means for demonstrating the correctness of the analysis, but they also provide some intuition into the use of the analysis. In a more practical implementation of a functional language we expect this analysis to provide useful information for the implementation of some function calls and also in the design of function closures.

Acknowledgements

This work was supported in part by an NSF CAREER award. Patricia Burns and Patrick Hicks assisted in the proof of Theorem 5.4 and provided valuable comments and criticisms. Martin Ibarra encoded many of the proofs into the Elf programming language. Aleksandar Djakovic and Mads Tofte contributed with valuable discussions, and Alexander Smart provided support and encouragement. The anonymous referees made several useful comments and suggestions.

9 Proof of Theorem 5.4

The following propositions provide useful properties for manipulating the sets Δ in consistency relations:

Proposition 9.1
If $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ then $\Delta \subseteq \mathsf{dom}(\Gamma)$.

The proof follows by induction on the definition of consistency. Observe that this proposition implies that $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ and $x^* \notin \mathsf{dom}(\Gamma)$ imply $x^* \notin \mathsf{dom}(\Delta)$. We use this observation in the proof of Lemma 9.6 below.

Proposition 9.2 (Weakening)

1. If $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ and $\Delta' \subseteq \Delta$ then $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta')$.
2. If $\mathsf{Consistent}(\pi, v, w, \phi, \Delta)$ and $\Delta' \subseteq \Delta$ then $\mathsf{Consistent}(\pi, v, w, \phi, \Delta')$.

The proof follows by induction on the definition of consistency. Note that for each variable $x^* \in \Delta$, the construction of $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ requires one application of rule (6) or (7). Hence for each variable in $x^* \in \Delta - \Delta'$ we must replace the application of rule (6) or (7) (which requires $x^* \in \Delta'$) with applications of rules (8) and (9) which has the combined net effect of rule (6) or (7), except that they do not require $x^* \notin \Delta'$.

We need to demonstrate that values extracted from a stack or an environment are consistent with a larger stack. The following propositions contribute towards this goal.

Proposition 9.3
If $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ then

1. $\mathsf{dom}(\rho) = |\mathsf{dom}(\Gamma)|$;
2. $\mathsf{Distinct}(\rho)$ implies $\mathsf{Distinct}(\Gamma)$.

The proof follows by induction on the definition of consistency.

Proposition 9.4
If $\mathsf{LV}(\phi) \cap \Delta_1 = \mathsf{LV}(\phi) \cap \Delta_2$ then

$$\mathsf{Consistent}(\pi, v, w, \phi, \Delta_1) \text{ iff } \mathsf{Consistent}(\pi, v, w, \phi, \Delta_2).$$

The proof follows directly from rules 1–3 of Definition 5.3.

We often need to account for the consistency of two values before and after popping the stack. The following provides a suitable property for doing so:

Proposition 9.5
If $x^* \notin (\mathsf{LV}(\phi) \cap \Delta)$ then

$$\mathsf{Consistent}(\pi, v, w, \phi, \Delta - \{x^*\}) \text{ iff } \mathsf{Consistent}(\pi\{x^* \mapsto w'\}, v, w, \phi, \Delta).$$

Proof
In the forward direction we have two cases based on the structure of the values. If $v = w = c$ for some constant c then the statement holds from the definition of value consistency. Otherwise, v and w are closures. We consider just the case in which v and w are non-recursive closures; the cases for closures involving recursive functions follow similarly. Assume $v = [\rho, e]$ and $w = [\eta, m]$ for some ρ, e, η, m. Then from $\mathsf{Consistent}(\pi, [\rho, e], [\eta, m], \phi, \Delta - \{x^*\})$ we have that there exists a Γ such that

$$\Gamma \rhd e : (\phi, \{\}) \Rightarrow m, \tag{1}$$

$$\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, (\Delta - \{x^*\} \cap \mathsf{LV}(\phi))). \tag{2}$$

Because $x^* \notin (\Delta \cap \mathsf{LV}(\phi))$ we have $(\Delta - \{x^*\}) \cap \mathsf{LV}(\phi) = \Delta \cap \mathsf{LV}(\phi)$. Let $\Delta' = \Delta \cap \mathsf{LV}(\phi)$. To show $\mathsf{Consistent}(\pi\{x^* \mapsto w'\}, [\rho, e], [\eta, m], \phi, \Delta)$ we use item (1) and need only show

$$\mathsf{Consistent}(\rho, \pi\{x^* \mapsto w'\}, \eta, \Gamma, \Delta'). \tag{3}$$

This follows from (2) using rule (9) of the definition of consistency.

For the reverse direction we again have two cases. If $v = w = c$ for some constant c then the statement follows from the definition of consistency for constants. Otherwise v and w are closures. We consider just the case in which v and w are non-recursive closures; the cases for closures involving recursive functions follow similarly. Assume $v = [\rho, e]$ and $w = [\eta, m]$. From the assumption

$$\mathsf{Consistent}(\pi\{x^*\mapsto w'\}, [\rho, e], [\eta, m], \phi, \Delta) \tag{4}$$

we have that there exists a Γ such that

$$\Gamma \rhd e : (\phi, \{\}) \Rightarrow m, \tag{5}$$

$$\mathsf{Consistent}(\rho, \pi\{x^*\mapsto w'\}, \eta, \Gamma, \Delta \cap \mathsf{LV}(\phi)). \tag{6}$$

To show $\mathsf{Consistent}(\pi, [\rho, e], [\eta, m], \phi, \Delta - \{x^*\})$ we use item (5) above and then need only establish

$$\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, (\Delta - \{x^*\} \cap \mathsf{LV}(\phi))). \tag{7}$$

Again, because $x^* \notin (\Delta \cap \mathsf{LV}(\phi))$ we have $(\Delta - \{x^*\}) \cap \mathsf{LV}(\phi) = \Delta \cap \mathsf{LV}(\phi)$. Let $\Delta' = \Delta \cap \mathsf{LV}(\phi)$. Note $x^* \notin \Delta'$. Then

$$\mathsf{Consistent}(\rho, \pi\{x^*\mapsto w'\}, \eta, \Gamma, \Delta') \tag{8}$$

must have followed inductively (via some number - 0 or more - of applications of rules (5) and (8) of Definition 5.3) from

$$\mathsf{Consistent}(\rho', \pi\{x^*\mapsto w'\}, \eta', \Gamma', \Delta') \tag{9}$$

for some ρ', η' and Γ' such that $\rho' \sqsubseteq \rho$, $\eta' \sqsubseteq \eta$ and $\Gamma' \sqsubseteq \Gamma$, which, in turn, must be the result of an application of rule (9) (because $x^* \notin \Delta'$) from

$$\mathsf{Consistent}(\rho', \pi, \eta', \Gamma', \Delta'). \tag{10}$$

From this we can then “reapply” the sequence of rules (5) and (8) from above to obtain

$$\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta'). \tag{11}$$

□

A required property of consistency of environments and stacks is that a variable map to related values in the source and target languages. We have the following lemma to ensure this.

Lemma 9.6 (*Variable Consistency*)
If $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ and $\mathsf{Distinct}(\rho)$ then:

1. if $(y : \phi) \in \Gamma$ then $y \in \mathsf{dom}(\rho)$, $y \in \mathsf{dom}(\eta)$, and $\mathsf{Consistent}(\pi, \rho(y), \eta(y), \phi, \Delta)$;
2. if $(y^* {:} \phi) \in \Gamma$ and $y^* \in \Delta$ then $y \in \mathsf{dom}(\rho)$, $y^* \in \mathsf{dom}(\pi)$, and $\mathsf{Consistent}(\pi, \rho(y), \pi(y^*), \phi, \Delta)$.

Proof
The proof for both parts proceeds by induction on the definition of consistency. For the base case (rule (4)) both parts hold vacuously. We consider the inductive steps for each case separately. The proof of each decomposes into five cases, corresponding to rules (5) through (9) of the definition of consistency. We use the property that $\mathsf{Distinct}(\rho)$ implies $\mathsf{Distinct}(\Gamma)$.
Inductive step for part 1:

Rule 5: Assume Consistent($\rho\{x{\mapsto}v\}, \pi, \eta\{x{\mapsto}w\}, \Gamma\{x : \phi'\}, \Delta$) because (LV($\phi'$)$\cap\Delta$) $\subseteq$ dom(Γ), Consistent($\rho, \pi, \eta, \Gamma, \Delta$) and Consistent($\pi, v, w, \phi', \Delta$). If $x = y$ then $\phi = \phi'$ and the statement holds from Consistent(π, v, w, ϕ, Δ). If $x \neq y$ then by induction on Consistent($\rho, \pi, \eta, \Gamma, \Delta$) we have Consistent($\pi, \rho(y), \eta(y), \phi, \Delta$), which, along with $x \neq y$ implies Consistent($\pi, \rho\{x{\mapsto}v\}(y), \eta\{x{\mapsto}w\}(y), \phi, \Delta$).

Rule 6: Assume Consistent($\rho\{x{\mapsto}v\}, \pi\{x^*{\mapsto}w\}, \eta, \Gamma\{x^* : \phi'\}, \Delta \uplus \{x^*\}$) because (LV($\phi'$)$\cap\Delta$) $\subseteq$ dom(Γ), Consistent($\rho, \pi, \eta, \Gamma, \Delta$) and Consistent($\pi, v, w, \phi', \Delta$). Then $x \neq y$ (by Distinct($\Gamma\{x^* : \phi'\}$)), and by induction on Consistent($\rho, \pi, \eta, \Gamma, \Delta$) we have Consistent($\pi, \rho(y), \eta(y), \phi, \Delta$). We have $x^* \notin \Delta$, and so by Proposition 9.5 we then have Consistent($\pi\{x^*{\mapsto}w\}, \rho(y), \eta(y), \phi, \Delta \uplus \{x^*\}$), which along with $x \neq y$ implies Consistent($\pi\{x^*{\mapsto}w\}, \rho\{x{\mapsto}v\}(y), \eta(y), \phi, \Delta \uplus \{x^*\}$).

Rule 7: Assume Consistent($\rho\{f{\mapsto}\left[\rho', \mu f.\lambda x.e\right]\}, \pi\{f^*{\mapsto}\left[\eta', \lambda x^*.m\right]\}, \eta, \Gamma\{f^* : \phi'\}, \Delta\uplus \{f^*\}$) because (LV($\phi'$) $\cap \Delta$) $\subseteq$ dom(Γ), Consistent($\rho, \pi, \eta, \Gamma, \Delta$) and Consistent($\pi, \left[\rho', \mu f.\lambda x.e\right], \left[\eta', \mu f^*.\lambda x^*.m\right], \phi', \Delta$). Then $f \neq y$ (by Distinct($\Gamma\{f^* : \phi'\}$)), and by induction on Consistent($\rho, \pi, \eta, \Gamma, \Delta$) we have Consistent($\pi, \rho(y), \eta(y), \phi, \Delta$). We have $f^* \notin \Delta$, and so by Proposition 9.5 we then have Consistent($\pi\{f^*{\mapsto}\left[\eta', \lambda x^*.m\right]\}, \rho(y), \eta(y), \phi, \Delta\uplus\{f^*\}$), which along with $f{\neq}y$ implies Consistent($\pi\{f^*{\mapsto}\left[\eta', \lambda x^*.m\right]\}, \rho\{f{\mapsto}\left[\rho', \mu f.\lambda x.e\right]\}(y), \eta(y), \phi, \Delta \uplus \{f^*\}$).

Rule 8: Assume Consistent($\rho\{x{\mapsto}v\}, \pi, \eta, \Gamma\{x^* : \phi'\}, \Delta$) because (LV($\phi'$) $\cap \Delta$) $\subseteq$ dom(Γ), and Consistent($\rho, \pi, \eta, \Gamma, \Delta$). Then $x \neq y$ (by Distinct($\Gamma\{x^* : \phi'\}$)) and by induction we have Consistent($\pi, \rho(y), \eta(y), \phi, \Delta$). This and $x \neq y$ imply Consistent($\pi, \rho\{x{\mapsto}v\}(y), \eta(y), \phi, \Delta$).

Rule 9: Assume Consistent($\rho, \pi\{x^*{\mapsto}w'\}, \eta, \Gamma, \Delta$) because $x^* \notin \Delta$, and Consistent($\rho, \pi, \eta, \Gamma, \Delta$). Then by induction we have Consistent($\pi, \rho(y), \eta(y), \phi, \Delta$). We have $x \notin$ LV(ϕ)$\cap\Delta$ and so by Proposition 9.5 we then also have Consistent($\pi\{x^*{\mapsto}w'\}, \rho(y), \eta(y), \phi, \Delta$) (because $\Delta - \{x^*\} = \Delta$).

Inductive step for part 2:

Rule 5: Assume Consistent($\rho\{x{\mapsto}v\}, \pi, \eta\{x{\mapsto}w\}, \Gamma\{x : \phi'\}, \Delta$) because (LV($\phi'$) $\cap \Delta$) $\subseteq$ dom(Γ), Consistent($\rho, \pi, \eta, \Gamma, \Delta$), and Consistent($\pi, v, w, \phi', \Delta$). Then $x \neq y$ (by Distinct($\Gamma\{x : \phi'\}$)) and by induction on Consistent($\rho, \pi, \eta, \Gamma, \Delta$) we have Consistent($\pi, \rho(y), \pi(y^*), \phi, \Delta$). This and $x \neq y$ imply Consistent($\pi, \rho\{x{\mapsto}v\}(y), \pi(y^*), \phi, \Delta$).

Rule 6: Assume Consistent($\rho\{x{\mapsto}v\}, \pi\{x^*{\mapsto}w\}, \eta, \Gamma\{x^* : \phi'\}, \Delta \uplus \{x^*\}$) because (LV($\phi'$) $\cap \Delta$) $\subseteq$ dom(Γ), Consistent($\rho, \pi, \eta, \Gamma, \Delta$) and Consistent($\pi, v, w, \phi', \Delta$). If $x = y$ then $\phi = \phi'$ and we have Consistent(π, v, w, ϕ, Δ). we have $x^* \notin \Delta$, and so by Proposition 9.5 we have Consistent($\pi\{x^*{\mapsto}w\}, v, w, \phi, \Delta \uplus \{x^*\}$).
If $x \neq y$ then by induction on Consistent($\rho, \pi, \eta, \Gamma, \Delta$) we have Consistent($\pi, \rho(y), \pi(y^*), \phi, \Delta$). We have $x^* \notin \Delta$ and so by Proposition 9.5 we then have Consistent($\pi\{x^*{\mapsto}w\}, \rho(y), \pi(y^*), \phi, \Delta \uplus \{x^*\}$). This and $x \neq y$ imply Consistent ($\pi\{x^*{\mapsto}w\}, \rho\{x{\mapsto}v\}(y), \pi\{x^*{\mapsto}w\}(y^*), \phi, \Delta \uplus \{x^*\}$).

Rule 7: Assume Consistent($\rho\{f{\mapsto}\left[\rho', \mu f.\lambda x.e\right]\}, \pi\{f^*{\mapsto}\left[\eta', \lambda x^*.m\right]\}, \eta, \Gamma\{f^* : \phi'\}, \Delta\uplus \{f^*\}$) because (LV($\phi'$) $\cap \Delta$) $\subseteq$ dom(Γ), Consistent($\rho, \pi, \eta, \Gamma, \Delta$) and Consistent($\pi$,

$\left[\rho', \mu f.\lambda x.e\right], \left[\eta', \mu f^*.\lambda x^*.m\right], \phi', \Delta)$. If $f = y$ then $\phi = \phi'$ and we have

$$\mathsf{Consistent}(\rho\{f \mapsto \left[\rho', \mu f.\lambda x.e\right]\}, \pi\{f^* \mapsto \left[\eta', \lambda x^*.m\right]\}, \eta, \Gamma\{f^* : \phi\}, \Delta \uplus \{f^*\}), \quad (12)$$

$$\mathsf{Consistent}(\pi, \left[\rho', \mu f.\lambda x.e\right], \left[\eta', \mu f^*.\lambda x^*.m\right], \phi, \Delta). \quad (13)$$

We need to show

$$\mathsf{Consistent}(\pi\{f^* \mapsto \left[\eta', \lambda x^*.m\right]\}, \left[\rho', \mu f.\lambda x.e\right], \left[\eta', \lambda x^*.m\right], \phi, \Delta \uplus \{f^*\}), \quad (14)$$

which by Definition 5.3 requires showing that there exists a Γ' such that

$$\Gamma' \rhd \mu f.\lambda x.e : (\phi, \{\}) \Rightarrow \mu f^*.\lambda x^*.m, \quad (15)$$

$$\begin{aligned}&\mathsf{Consistent}(\rho'\{f \mapsto \left[\rho', \mu f.\lambda x.e\right]\}, \pi\{f^* \mapsto \left[\eta', \lambda x^*.m\right]\}, \\ &\quad \eta', \Gamma'\{f^* : \phi\}, ((\Delta \uplus \{f^*\}) \cap \mathsf{LV}(\phi)) \uplus \{f^*\}).\end{aligned} \quad (16)$$

Observe that $((\Delta \uplus \{f^*\}) \cap \mathsf{LV}(\phi)) \uplus \{f^*\} = (\Delta \cap \mathsf{LV}(\phi)) \uplus \{f^*\}$. Item (13) implies that we have a Γ' such that item (15) holds and

$$\mathsf{Consistent}(\rho', \pi, \eta', \Gamma', (\Delta \cap \mathsf{LV}(\phi))) \quad (17)$$

By Proposition 9.1, this implies $(\Delta \cap \mathsf{LV}(\phi)) \subset \Gamma'$. Applying Proposition 9.2 to item (13) yields

$$\mathsf{Consistent}(\pi, \left[\rho', \mu f.\lambda x.e\right], \left[\eta', \mu f^*.\lambda x^*.m\right], \phi, \Delta \cap \mathsf{LV}(\phi)). \quad (18)$$

Using $(\Delta \cap \mathsf{LV}(\phi)) \subset \Gamma'$ and items (17) and (18) we can apply rule (7) of Definition 5.3 to obtain (16).

If $f \neq y$ then by induction on $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ we have $\mathsf{Consistent}(\pi, \rho(y), \pi(y^*), \phi, \Delta)$. We have $f^* \notin \Delta$ and so by Proposition 9.5 we then have $\mathsf{Consistent}(\pi\{f^* \mapsto \left[\eta', \lambda x^*.m\right]\}, \rho(y), \pi(y^*), \phi, \Delta \uplus \{f^*\})$. This and $f \neq y$ imply $\mathsf{Consistent}(\pi\{f^* \mapsto \left[\eta', \lambda x^*.m\right]\}, \rho\{f \mapsto \left[\rho', \mu f.\lambda x.e\right]\}(y), \pi\{f^* \mapsto \left[\eta', \lambda x^*.m\right]\}(y^*), \phi, \Delta \uplus \{f^*\})$.

Rule 8: Assume $\mathsf{Consistent}(\rho\{x \mapsto v\}, \pi, \eta, \Gamma\{x^* : \phi'\}, \Delta)$ because $(\mathsf{LV}(\phi') \cap \Delta) \subseteq \mathsf{dom}(\Gamma)$, and $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$. Then $x \neq y$ (because $x^* \notin \Delta$ which we obtain from $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$ and $x^* \notin \mathsf{dom}(\Gamma)$ by Proposition 9.1) and by induction we have $\mathsf{Consistent}(\pi, \rho(y), \pi(y^*), \phi, \Delta)$. This and $x \neq y$ imply $\mathsf{Consistent}(\pi, \rho\{x \mapsto v\}(y), \pi(y^*), \phi, \Delta)$.

Rule 9: Assume $\mathsf{Consistent}(\rho, \pi\{x^* \mapsto w\}, \eta, \Gamma, \Delta)$ because $x^* \notin \Delta$ and $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta)$. Then $x \neq y$ (because $x^* \notin \Delta$) and by induction we have $\mathsf{Consistent}(\pi, \rho(y), \pi(y^*), \phi, \Delta)$. Observe that $x^* \notin (\mathsf{LV}(\phi) \cap \Delta)$. By Proposition 9.5 we then also have $\mathsf{Consistent}(\pi\{x^* \mapsto w\}, \rho(y), \pi(y^*), \phi, \Delta)$ (because $\Delta - \{x^*\} = \Delta$). And because $x \neq y$ we also have $\mathsf{Consistent}(\pi\{x^* \mapsto w\}, \rho(y), \pi\{x^* \mapsto w\}(y^*), \phi, \Delta)$.

□

Theorem 5.4

If $\mathsf{Distinct}([\rho, e])$, $\Xi :: \Gamma \rhd e : (\phi, \Delta_0) \Rightarrow m$, $\Delta_0 \subseteq \Delta'$, and $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta')$ then

1. $\Pi :: \rho \rhd e \hookrightarrow v$ implies that there exist w and $\Theta :: \pi; \eta \rhd m \hookrightarrow_t w$ such that $\mathsf{Consistent}(\pi, v, w, \phi, \Delta')$;

2. $\Theta :: \pi;\eta \rhd m \hookrightarrow_t w$ implies that there exist v and $\Pi :: \rho \rhd e \hookrightarrow v$ such that $\mathsf{Consistent}(\pi, v, w, \phi, \Delta')$.

Proof

We prove only the first part; the second follows similarly. The proof follows by induction on Π. We assume $\mathsf{Distinct}([\rho, e])$, $\Xi :: \Gamma \rhd e : (\phi, \Delta_0) \Rightarrow m$, $\Delta_0 \subseteq \Delta'$, $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta')$, and $\Pi :: \rho \rhd e \hookrightarrow v$. We show that there exist w and $\Theta :: \pi;\eta \rhd m \hookrightarrow_t w$ such that $\mathsf{Consistent}(\pi, v, w, \phi, \Delta')$.

1. Π is $\dfrac{}{\rho \rhd c \hookrightarrow c}$ and Ξ is $\dfrac{(c : |\phi|) \in \Sigma}{\Gamma \rhd c : (\phi, \{\}) \Rightarrow c}$. Then Θ is $\dfrac{}{\pi;\eta \rhd c \hookrightarrow_t c}$ and we have $\mathsf{Consistent}(\pi, c, c, \phi, \Delta')$.
2. Π is $\dfrac{\rho(x) = v}{\rho \rhd x \hookrightarrow v}$. Then we have two possible cases for the structure of Ξ:

   (a) Ξ is the deduction $\dfrac{(x : \phi) \in \Gamma}{\Gamma \rhd x : (\phi, \{\}) \Rightarrow x}$. By Lemma 9.6 we have $x \in \mathsf{dom}(\eta)$ and $\mathsf{Consistent}(\pi, \rho(x), \eta(x), \phi, \Delta')$. Θ is then of the form

   $$\frac{\eta(x) = w}{\pi;\eta \rhd x \hookrightarrow w}.$$

   (b) Ξ is the deduction $\dfrac{(x^* : \phi) \in \Gamma}{\Gamma \rhd x : (\phi, \{x^*\}) \Rightarrow x^*}$. We have $\Delta_0 = \{x^*\}$ and hence $x^* \in \Delta'$ by assumption. By Lemma 9.6 we have $x^* \in \mathsf{dom}(\pi)$ and $\mathsf{Consistent}(\pi, \rho(x), \pi(x^*), \phi, \Delta')$. Then Θ is of the form

   $$\frac{\pi(x^*) = w}{\pi;\eta \rhd x^* \hookrightarrow w}.$$

3. Assume Π is of the form

   $$\frac{\begin{array}{cc}\Pi_1 & \Pi_2\\ \rho \rhd e_1 \hookrightarrow \mathsf{true} & \rho \rhd e_2 \hookrightarrow v\end{array}}{\rho \rhd \mathsf{if}\ m_1\ m_2\ m_3 \hookrightarrow v}.$$

   (The case in which e_1 evaluates to false follows similarly.) Then Ξ is of the form

   $$\frac{\begin{array}{ccc}\Xi_1 & \Xi_2 & \Xi_3\\ \Gamma \rhd e_1 : (\mathsf{bool}, \Delta_1) \Rightarrow m_1 & \Gamma \rhd e_2 : (\phi, \Delta_2) \Rightarrow m_2 & \Gamma \rhd e_3 : (\phi, \Delta_3) \Rightarrow m_3\end{array}}{\Gamma \rhd \mathsf{if}\ e_1\ e_2\ e_3 : (\phi, \Delta_1 \cup \Delta_2 \cup \Delta_3) \Rightarrow \mathsf{if}\ m_1\ m_2\ m_3}.$$

   Note that $\Delta_0 = \Delta_1 \cup \Delta_2 \cup \Delta_3$. Applying the inductive hypothesis to Π_1 and Π_2 (using Ξ_1, Ξ_2, $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta')$, $\mathsf{Distinct}([\rho, e_1])$, $\mathsf{Distinct}([\rho, e_2])$, $\Delta_1 \subseteq \Delta'$, and $\Delta_2 \subseteq \Delta'$) we have

   $$\Theta_1 :: \pi;\eta \rhd m_1 \hookrightarrow_t \mathsf{true}, \quad (19)$$

   $$\Theta_2 :: \pi;\eta \rhd m_2 \hookrightarrow_t w, \quad (20)$$

   $$\mathsf{Consistent}(\pi, \mathsf{true}, \mathsf{true}, \mathsf{bool}, \Delta'), \quad (21)$$

   $$\mathsf{Consistent}(\pi, v, w, \phi_1, \Delta'). \quad (22)$$

Then we can construct Θ as

$$\frac{\overset{\Theta_1}{\pi;\eta \ \rhd m_1 \hookrightarrow_t \mathsf{true}} \qquad \overset{\Theta_2}{\pi;\eta \ \rhd m_2 \hookrightarrow_t w}}{\pi;\eta \ \rhd \mathsf{if}\ m_1\ m_2\ m_3 \hookrightarrow_t w}.$$

4. Π is $\dfrac{}{\rho \ \rhd \lambda x.e_1 \hookrightarrow [\rho, \lambda x.e_1]}$. Then Ξ is a deduction of $\Gamma \ \rhd \lambda x.e_1 : (\phi, \{\}) \Rightarrow \lambda z.m_1$ where z is either x or x^*. Then Θ is $\dfrac{}{\pi;\eta \ \rhd \lambda z.m_1 \hookrightarrow_t [\eta, \lambda z.m_1]}$. To show $\mathsf{Consistent}(\pi, [\rho, \lambda x.e_1], [\eta, \lambda z.m_1], \phi, \Delta')$ we use Ξ and need only show $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma', \Delta' \cap \mathsf{LV}(\phi))$ which we obtain by applying Proposition 9.2 to $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma', \Delta')$.
   The case for recursive functions follows similarly.
5. Π is of the form

$$\frac{\overset{\Pi_1}{\rho \ \rhd e_1 \hookrightarrow \left[\rho', \lambda x.e'\right]} \qquad \overset{\Pi_2}{\rho \ \rhd e_2 \hookrightarrow v_2} \qquad \overset{\Pi_3}{\rho'\{x \mapsto v_2\} \ \rhd e' \hookrightarrow v}}{\rho \ \rhd (e_1 @ e_2) \hookrightarrow v}.$$

From $\mathsf{Distinct}([\rho, (e_1 @ e_2)])$ we have $\mathsf{Distinct}([\rho, e_1])$, $\mathsf{Distinct}([\rho, e_2])$, and from Lemma 5.2 and Definition 5.1 we have $\mathsf{Distinct}(\left[\rho'\{x \mapsto v_2\}, e'\right])$. Then we have two possible cases for the structure of Ξ:

(a) Ξ is a deduction of

$$\frac{\overset{\Xi_1}{\Gamma \ \rhd e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1) \Rightarrow m_1} \qquad \overset{\Xi_2}{\Gamma \ \rhd e_2 : (\phi_1, \Delta_2) \Rightarrow m_2}}{\Gamma \ \rhd (e_1 @ e_2) : (\phi_2, \Delta \cup \Delta_1 \cup \Delta_2) \Rightarrow (m_1 @ m_2)}.$$

Note that $\Delta_0 = \Delta \cup \Delta_1 \cup \Delta_2$. Applying the inductive hypothesis to Π_1 and Π_2 (using Ξ_1, Ξ_2, $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta')$, $\mathsf{Distinct}([\rho, e_1])$, $\mathsf{Distinct}([\rho, e_2])$, $\Delta_1 \subseteq \Delta'$, and $\Delta_2 \subseteq \Delta'$), we have

$$\Theta_1 :: \pi;\eta \ \rhd m_1 \hookrightarrow_t \left[\eta', \lambda x.m'\right], \tag{23}$$
$$\Theta_2 :: \pi;\eta;\ \rhd m_2 \hookrightarrow_t w_2, \tag{24}$$
$$\mathsf{Consistent}(\pi, \left[\rho', \lambda x.e'\right], \left[\eta', \lambda x.m'\right], \phi_1 \xrightarrow{\Delta} \phi_2, \Delta'), \tag{25}$$
$$\mathsf{Consistent}(\pi, v_2, w_2, \phi_1, \Delta'). \tag{26}$$

Let $\Delta_a = \Delta' \cap \mathsf{LV}(\phi_1 \xrightarrow{\Delta} \phi_2)$. Item (25) implies that there exists some Γ' such that

$$\Gamma' \ \rhd \lambda x.e' : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\}) \Rightarrow \lambda x.m', \tag{27}$$
$$\mathsf{Consistent}(\rho', \pi, \eta', \Gamma', \Delta_a). \tag{28}$$

Applying Proposition 9.1 to item 28 yields

$$\Delta_a \subseteq \mathsf{dom}(\Gamma'). \tag{29}$$

Applying Proposition 9.2 to item (26) and $\Delta_a \subseteq \Delta'$ yields

$$\mathsf{Consistent}(\pi, v_2, w_2, \phi_1, \Delta_a). \tag{30}$$

Item (27) implies that we have a derivation of

$$\Gamma'\{x : \phi_1\} \rhd e' : (\phi_2, \Delta'') \Rightarrow m' \tag{31}$$

in which $\Delta'' \subseteq \Delta$. Using $(\mathsf{LV}(\phi_1) \cap \Delta_a) \subseteq \mathsf{dom}(\Gamma')$ (which follows from item (29)) and items (28) and (30), we can apply rule (5) of Definition 5.3 to obtain

$$\mathsf{Consistent}(\rho'\{x \mapsto v_2\}, \pi, \eta'\{x \mapsto w_2\}, \Gamma'\{x : \phi_1\}, \Delta_a). \tag{32}$$

By some simple reasoning we have $\Delta'' \subseteq \Delta \subseteq \Delta_a$. Using this property, items (32) and (31), and the property $\mathsf{Distinct}(\left[\rho'\{x \mapsto v_2\}, e'\right])$, we can apply the inductive hypothesis to Π_3, yielding that there exists some w and Θ_3 such that

$$\Theta_3 :: \pi ; \eta'\{x \mapsto w_2\} \rhd m' \hookrightarrow_t w, \tag{33}$$

$$\mathsf{Consistent}(\pi, v, w, \phi_2, \Delta_a). \tag{34}$$

By some simple reasoning we have $\Delta_a \cap \mathsf{LV}(\phi_2) = \Delta' \cap \mathsf{LV}(\phi_2)$. From this and item (34), Proposition 9.4 yields

$$\mathsf{Consistent}(\pi, v, w, \phi_2, \Delta'). \tag{35}$$

Finally we can construct the deduction Θ as:

$$\frac{\overset{\Theta_1}{\pi ; \eta \rhd m_1 \hookrightarrow_t [\eta', \lambda x.m']} \qquad \overset{\Theta_2}{\pi ; \eta \rhd m_2 \hookrightarrow_t w_2} \qquad \overset{\Theta_3}{\pi ; \eta'\{x \mapsto w_2\} \rhd m' \hookrightarrow_t w}}{\pi ; \eta \rhd (m_1 \mathbin{@} m_2) \hookrightarrow_t w}.$$

(b) Ξ is a deduction of

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta}_* \phi_2, \Delta_1) \Rightarrow m_1} \qquad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi_1, \Delta_2) \Rightarrow m_2}}{\Gamma \rhd (e_1 \mathbin{@} e_2) : (\phi_2, \Delta \cup \Delta_1 \cup \Delta_2) \Rightarrow (m_1 \mathbin{@}^* m_2)}.$$

Note that $\Delta_0 = \Delta \cup \Delta_1 \cup \Delta_2$. Applying the inductive hypothesis to Π_1 and Π_2 (using Ξ_1, Ξ_2, $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta')$, $\mathsf{Distinct}([\rho, e_1])$, $\mathsf{Distinct}([\rho, e_2])$, $\Delta_1 \subseteq \Delta'$, and $\Delta_2 \subseteq \Delta'$), we have

$$\Theta_1 :: \pi ; \eta \rhd m_1 \hookrightarrow_t \left[\eta', \lambda x^*.m'\right], \tag{36}$$

$$\Theta_2 :: \pi ; \eta ; \rhd m_2 \hookrightarrow_t w_2, \tag{37}$$

$$\mathsf{Consistent}(\pi, \left[\rho', \lambda x.e'\right], \left[\eta', \lambda x^*.m'\right], \phi_1 \xrightarrow{\Delta}_* \phi_2, \Delta'), \tag{38}$$

$$\mathsf{Consistent}(\pi, v_2, w_2, \phi_1, \Delta'). \tag{39}$$

Let $\Delta_a = \Delta' \cap \mathsf{LV}(\phi_1 \xrightarrow{\Delta}_* \phi_2)$. Item (38) implies that there exists some Γ' such that

$$\Gamma' \rhd \lambda x.e' : (\phi_1 \xrightarrow{\Delta}_* \phi_2, \{\}) \Rightarrow \lambda x^*.m', \tag{40}$$

$$\mathsf{Consistent}(\rho', \pi, \eta', \Gamma', \Delta_a). \tag{41}$$

From (41) and $\mathsf{Distinct}(\left[\rho'\{x \mapsto v_2\}, e'\right])$, Proposition 9.3 implies $x^* \notin \mathsf{dom}(\Gamma')$, Applying Proposition 9.1 to item 41 yields

$$\Delta_a \subseteq \mathsf{dom}(\Gamma'). \tag{42}$$

This and $x^* \notin \mathsf{dom}(\Gamma')$ yields $x^* \notin \Delta_a$. Applying Proposition 9.2 to item (39) and $\Delta_a \subseteq \Delta'$ yields

$$\mathsf{Consistent}(\pi, v_2, w_2, \phi_1, \Delta_a). \tag{43}$$

Item (40) implies that we have a derivation of

$$\Gamma'\{x^* : \phi_1\} \rhd e' : (\phi_2, \Delta'') \Rightarrow m' \tag{44}$$

in which $\Delta'' \subseteq \Delta \uplus \{x^*\}$ and $x^* \notin \mathsf{LV}(\phi_2)$. Using $(\mathsf{LV}(\phi_1) \cap \Delta_a) \subseteq \mathsf{dom}(\Gamma')$ (which follows from item (42)) and items (41) and (43), we can apply rule (6) of Definition 5.3 to obtain

$$\mathsf{Consistent}(\rho'\{x \mapsto v_2\}, \pi\{x^* \mapsto w_2\}, \eta', \Gamma'\{x^* : \phi_1\}, \Delta_a \uplus \{x^*\}). \tag{45}$$

Observe that $\Delta'' \subseteq \Delta \uplus \{x^*\} \subseteq (\Delta_a \uplus \{x^*\})$. Using this property, items (45) and (44), and the property $\mathsf{Distinct}(\left[\rho'\{x \mapsto v_2\}, e'\right])$, we can apply the inductive hypothesis to Π_3, yielding that there exists some w and Θ_3 such that

$$\Theta_3 :: \pi\{x^* \mapsto w_2\}; \eta' \rhd m' \hookrightarrow_t w, \tag{46}$$

$$\mathsf{Consistent}(\pi\{x^* \mapsto w_2\}, v, w, \phi_2, \Delta_a \uplus \{x^*\}). \tag{47}$$

By some simple reasoning we have $(\Delta_a \uplus \{x^*\}) \cap \mathsf{LV}(\phi_2) = (\Delta' \uplus \{x^*\}) \cap \mathsf{LV}(\phi_2)$. From this and item (47), Proposition 9.4 yields

$$\mathsf{Consistent}(\pi\{x^* \mapsto w_2\}, v, w, \phi_2, \Delta' \uplus \{x^*\}). \tag{48}$$

By Proposition 9.5, item (48) and $x^* \notin \mathsf{LV}(\phi_2)$ imply

$$\mathsf{Consistent}(\pi, v, w, \phi_2, \Delta'). \tag{49}$$

We can construct the deduction Θ as:

$$\frac{\overset{\Theta_1}{\pi;\eta \rhd m_1 \hookrightarrow_t [\eta', \lambda x^*.m']} \qquad \overset{\Theta_2}{\pi;\eta \rhd m_2 \hookrightarrow_t w_2} \qquad \overset{\Theta_3}{\pi\{x^* \mapsto w_2\};\eta' \rhd m' \hookrightarrow_t w}}{\pi;\eta \rhd (m_1 \mathbin{@^*} m_2) \hookrightarrow_t w}.$$

6. Π is of the form

$$\frac{\overset{\Pi_1}{\rho \rhd e_1 \hookrightarrow [\rho', \mu f.\lambda x.e']} \qquad \overset{\Pi_2}{\rho \rhd e_2 \hookrightarrow v_2} \qquad \overset{\Pi_3}{\rho'\{f \mapsto [\rho', \mu f.\lambda x.e']\}\{x \mapsto v_2\} \rhd e' \hookrightarrow v}}{\rho \rhd (e_1 \mathbin{@} e_2) \hookrightarrow v}$$

From $\mathsf{Distinct}([\rho, (e_1 \mathbin{@} e_2)])$ and Lemma 5.2 we have

$$\mathsf{Distinct}(\left[\rho'\{x \mapsto v_2\}\{f \mapsto \left[\rho', \mu f.\lambda x.e'\right]\}, e'\right]). \tag{50}$$

Then we have two possible cases for the structure of Ξ:

(a) Ξ is a deduction of

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1) \Rightarrow m_1} \qquad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi_1, \Delta_2) \Rightarrow m_2}}{\Gamma \rhd (e_1 \mathbin{@} e_2) : (\phi_2, \Delta \cup \Delta_1 \cup \Delta_2) \Rightarrow (m_1 \mathbin{@} m_2)}.$$

Note that $\Delta_0 = \Delta \cup \Delta_1 \cup \Delta_2$. Applying the inductive hypothesis to Π_1 and Π_2 (using Ξ_1, Ξ_2, $\mathsf{Consistent}(\rho, \pi, \eta, \Gamma, \Delta')$, $\mathsf{Distinct}([\rho, e_1])$, $\mathsf{Distinct}([\rho, e_2])$,

$\Delta_1 \subseteq \Delta'$, and $\Delta_2 \subseteq \Delta'$), we have

$$\Theta_1 :: \pi ; \eta \rhd m_1 \hookrightarrow_t \left[\eta', \mu f.\lambda x.m'\right], \tag{51}$$

$$\Theta_2 :: \pi ; \eta ; \rhd m_2 \hookrightarrow_t w_2, \tag{52}$$

$$\mathsf{Consistent}(\pi, \left[\rho', \mu f.\lambda x.e'\right], \left[\eta', \mu f.\lambda x.m'\right], \phi_1 \xrightarrow{\Delta} \phi_2, \Delta'), \tag{53}$$

$$\mathsf{Consistent}(\pi, v_2, w_2, \phi_1, \Delta'). \tag{54}$$

Let $\Delta_a = \Delta' \cap \mathsf{LV}(\phi_1 \xrightarrow{\Delta} \phi_2)$. Item (53) implies that there exists some Γ' such that

$$\Gamma' \rhd \mu f.\lambda x.e' : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\}) \Rightarrow \mu f.\lambda x.m', \tag{55}$$

$$\mathsf{Consistent}(\rho', \pi, \eta', \Gamma', \Delta_a). \tag{56}$$

Applying Proposition 9.1 to item 56 yields

$$\Delta_a \subseteq \mathsf{dom}(\Gamma'). \tag{57}$$

From $\Delta_a \subseteq \Delta'$ we can apply Proposition 9.2 to items (53) and (54) to yield

$$\mathsf{Consistent}(\pi, \left[\rho', \mu f.\lambda x.e'\right], \left[\eta', \mu f.\lambda x.m'\right], \phi_1 \xrightarrow{\Delta} \phi_2, \Delta_a), \tag{58}$$

$$\mathsf{Consistent}(\pi, v_2, w_2, \phi_1, \Delta_a). \tag{59}$$

Item (55) implies that we have a derivation of

$$\Gamma'\{f : \phi_1 \xrightarrow{\Delta} \phi_2\}\{x : \phi_1\} \rhd e' : (\phi_2, \Delta'') \Rightarrow m' \tag{60}$$

in which $\Delta'' \subseteq \Delta$. Using $(\mathsf{LV}(\phi_1 \xrightarrow{\Delta} \phi_2) \cap \Delta_a) \subseteq \mathsf{dom}(\Gamma')$ and $(\mathsf{LV}(\phi_1) \cap \Delta_a) \subseteq \mathsf{dom}(\Gamma')$ (which follow from item (57)) and items (56), (58), and (59), we can apply rule (5) of Definition 5.3 twice to obtain

$$\begin{aligned}&\mathsf{Consistent}(\rho'\{f \mapsto \left[\rho', \mu f.\lambda x.e'\right]\}\{x \mapsto v_2\}, \pi, \\ &\quad \eta'\{f \mapsto \left[\eta', \mu f.\lambda x.m'\right]\}\{x \mapsto w_2\}, \Gamma'\{f : \phi_1 \xrightarrow{\Delta} \phi_2\}\{x : \phi_1\}, \Delta_a.)\end{aligned} \tag{61}$$

Observe that $\Delta'' \subseteq \Delta \subseteq \Delta_a$. Using this property, items (61) and (60), and the property $\mathsf{Distinct}(\left[\rho'\{f \mapsto \left[\rho', \mu f.\lambda x.e'\right]\}\{x \mapsto v_2\}, e'\right])$, we can apply the inductive hypothesis to Π_3, yielding that there exists some w and Θ_3 such that

$$\Theta_3 :: \pi ; \eta'\{f \mapsto \left[\eta', \mu f.\lambda x.m'\right]\}\{x \mapsto w_2\} \rhd m' \hookrightarrow_t w, \tag{62}$$

$$\mathsf{Consistent}(\pi, v, w, \phi_2, \Delta_a). \tag{63}$$

With some simple reasoning we can show that $\Delta_a \cap \mathsf{LV}(\phi_2) = \Delta' \cap \mathsf{LV}(\phi_2)$. From this and item (63) Proposition 9.4 yields

$$\mathsf{Consistent}(\pi, v, w, \phi_2, \Delta'). \tag{64}$$

Finally we can construct the deduction Θ as:

$$\frac{\overset{\Theta_1}{\pi ; \eta \rhd m_1 \hookrightarrow_t [\eta', \mu f.\lambda x.m']} \qquad \overset{\Theta_2}{\pi ; \eta \rhd m_2 \hookrightarrow_t w_2} \qquad \overset{\Theta_3}{\pi ; \eta'\{f \mapsto [\eta', \mu f.\lambda x.m']\}\{x \mapsto w_2\} \rhd m' \hookrightarrow_t w}}{\pi ; \eta \rhd (m_1 \mathbin{@} m_2) \hookrightarrow_t w}.$$

(b) Ξ is a deduction of

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta}_* \phi_2, \Delta_1) \Rightarrow m_1} \qquad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi_1, \Delta_2) \Rightarrow m_2}}{\Gamma \rhd (e_1 @ e_2) : (\phi_2, \Delta \cup \Delta_1 \cup \Delta_2) \Rightarrow (m_1 @^* m_2)} .$$

Applying the inductive hypothesis to Π_1 and Π_2 (using Ξ_1, Ξ_2, Consistent $(\rho, \pi, \eta, \Gamma)\Delta'$, Distinct($[\rho, e_1]$), Distinct($[\rho, e_2]$), $\Delta_1 \subseteq \Delta'$, and $\Delta_2 \subseteq \Delta'$), we have

$$\Theta_1 :: \pi; \eta \rhd m_1 \hookrightarrow_t w_1, \qquad (65)$$

$$\Theta_2 :: \pi; \eta; \rhd m_2 \hookrightarrow_t w_2, \qquad (66)$$

$$\mathsf{Consistent}(\pi, [\rho', \mu f.\lambda x.e'], w_1, \phi_1 \xrightarrow{\Delta}_* \phi_2, \Delta'), \qquad (67)$$

$$\mathsf{Consistent}(\pi, v_2, w_2, \phi_1, \Delta'). \qquad (68)$$

Let $\Delta_a = \Delta' \cap \mathsf{LV}(\phi_1 \xrightarrow{\Delta}_* \phi_2)$. From item (67) and the definition of consistency we have two cases, based on the possible form of w_1:

(i) $w_1 = [\eta', \mu f^*.\lambda x^*.m']$ (Case 3a of Definition 5.3).
Item (67) implies that there exists some Γ' such that

$$\Gamma' \rhd \mu f.\lambda x.e' : (\phi_1 \xrightarrow{\Delta}_* \phi_2, \{\}) \Rightarrow \mu f^*.\lambda x^*.m', \qquad (69)$$

$$\mathsf{Consistent}(\rho', \pi, \eta', \Gamma', \Delta_a). \qquad (70)$$

From Distinct($[\rho'\{f \mapsto [\rho', \mu f.\lambda x.e']\}\{x \mapsto v_2\}, e']$), and item (70), Proposition 9.3 implies $x^* \notin \mathsf{dom}(\Gamma')$ and $f^* \notin \mathsf{dom}(\Gamma')$. Applying Proposition 9.1 to item (70) yields

$$\Delta_a \subseteq \mathsf{dom}(\Gamma'). \qquad (71)$$

Thus, $x^* \notin \mathsf{dom}(\Gamma')$ and $f^* \notin \mathsf{dom}(\Gamma')$ yield $x^* \notin \Delta_a$ and $f^* \notin \Delta_a$. From $\Delta_a \subseteq \Delta'$ we can apply Proposition 9.2 to items (67) and (68) to yield

$$\mathsf{Consistent}(\pi, [\rho', \mu f.\lambda x.e'], [\eta', \mu f^*.\lambda x^*.m'], \phi_1 \xrightarrow{\Delta}_* \phi_2, \Delta_a), \qquad (72)$$

$$\mathsf{Consistent}(\pi, v_2, w_2, \phi_1, \Delta_a). \qquad (73)$$

Item (69) implies that we have a derivation of

$$\Gamma'\{f^* : \phi_1 \xrightarrow{\Delta}_* \phi_2\}\{x^* : \phi_1\} \rhd e' : (\phi_2, \Delta'') \Rightarrow m' \qquad (74)$$

in which $\Delta'' \subseteq \Delta \uplus \{x^*, f^*\}$, $f^* \notin \mathsf{LV}(\phi_1)$ and $x^*, f^* \notin \mathsf{LV}(\phi_2)$. Using $(\mathsf{LV}(\phi_1 \xrightarrow{\Delta}_* \phi_2) \cap \Delta_a) \subseteq \mathsf{dom}(\Gamma')$ (which follows from item (71)) and items (70), and (72), we can apply rule (7) of Definition 5.3 to obtain

$$\mathsf{Consistent}(\rho'\{f \mapsto [\rho', \mu f.\lambda x.e']\}, \pi\{f^* \mapsto [\eta', \lambda x^*.m']\}, \qquad (75)$$
$$\eta', \Gamma'\{f^* : \phi_1 \xrightarrow{\Delta}_* \phi_2\}, \Delta_a \uplus \{f^*\}).$$

Using items (73) and the properties $f^* \notin \mathsf{LV}(\phi_1)$ and $f^* \notin \Delta_a$ we can apply Proposition 9.5 to obtain

$$\mathsf{Consistent}(\pi\{f^* \mapsto [\eta', \lambda x^*.m']\}, v_2, w_2, \phi_1, \Delta_a \uplus \{f^*\}). \qquad (76)$$

Using $(\mathsf{LV}(\phi_1) \cap (\Delta_a \uplus \{f^*\})) \subseteq \mathsf{dom}(\Gamma'\{f^* : \phi_1 \xrightarrow{\Delta}_* \phi_2\})$ (which follows from item (71)) and items (75) and (76), we can apply rule (6) of Definition 5.3 to obtain

$$\begin{aligned} &\mathsf{Consistent}(\rho'\{f \mapsto [\rho', \mu f.\lambda x.e']\}\{x \mapsto v_2\}, \\ &\quad \pi\{f^* \mapsto [\eta', \lambda x^*.m']\}\{x^* \mapsto w_2\}, \\ &\quad \eta', \Gamma'\{f^* : \phi_1 \xrightarrow{\Delta}_* \phi_2\}\{x^* : \phi_1\}, \Delta_a \uplus \{x^*, f^*\}). \end{aligned} \tag{77}$$

Observe that $\Delta'' \subseteq (\Delta \uplus \{x^*, f^*\}) \subseteq (\Delta_a \uplus \{x^*, f^*\})$. Using this property, items (50), (74) and (77), we can apply the inductive hypothesis to Π_3, yielding that there exists some w and Θ_3 such that

$$\Theta_3 :: \pi\{f^* \mapsto [\eta', \lambda x^*.m']\}\{x^* \mapsto w_2\}; \eta' \rhd m' \hookrightarrow_t w, \tag{78}$$

$$\mathsf{Consistent}(\pi\{f^* \mapsto [\eta', \lambda x^*.m']\}\{x^* \mapsto w_2\}, v, w, \phi_2, \Delta_a \uplus \{x^*, f^*\}). \tag{79}$$

By Proposition 9.5, item (79) and $x^*, f^* \notin \mathsf{LV}(\phi_2)$ imply

$$\mathsf{Consistent}(\pi, v, w, \phi_2, \Delta_a). \tag{80}$$

With some simple reasoning we can show that $\Delta_a \cap \mathsf{LV}(\phi_2) = \Delta' \cap \mathsf{LV}(\phi_2)$. From this and item (80), Proposition 9.4 yields

$$\mathsf{Consistent}(\pi, v, w, \phi_2, \Delta'). \tag{81}$$

Finally we can construct the deduction Θ from rule (4.10) and deductions Θ_1, Θ_2, and Θ_3.

(ii) $w_1 = [\eta', \lambda x^*.m']$. (Case 3b of Definition 5.3).
Item (67) implies that there exists some Γ' such that

$$\Gamma' \rhd \mu f.\lambda x.e' : (\phi_1 \xrightarrow{\Delta}_* \phi_2, \{\}) \Rightarrow \mu f^*.\lambda x^*.m', \tag{82}$$

$$\begin{aligned} &\mathsf{Consistent}(\rho'\{f \mapsto [\rho', \mu f.\lambda x.e']\}, \pi, \\ &\quad \eta', \Gamma'\{f^* : \phi_1 \xrightarrow{\Delta}_* \phi_2\}, \Delta_a \uplus \{f^*\}). \end{aligned} \tag{83}$$

From $\mathsf{Distinct}([\rho'\{f \mapsto [\rho', \mu f.\lambda x.e']\}\{x \mapsto v_2\}, e'])$, and item (70), Proposition 9.3 implies $x^* \notin \mathsf{dom}(\Gamma'\{f^* : \phi_1 \xrightarrow{\Delta}_* \phi_2\})$. Applying Proposition 9.1 to item (83) yields

$$(\Delta_a \uplus \{f^*\}) \subseteq \mathsf{dom}(\Gamma'\{f^* : \phi_1 \xrightarrow{\Delta}_* \phi_2\}). \tag{84}$$

Thus, $x^* \notin \mathsf{dom}(\Gamma')$ yields $x^* \notin \Delta_a$. (We already have $f^* \notin \Delta_a$ from $\Delta_a \uplus \{f^*\}$.) From $\Delta_a \subseteq \Delta'$ we can apply Proposition 9.2 to item (68) to yield

$$\mathsf{Consistent}(\pi, v_2, w_2, \phi_1, \Delta_a). \tag{85}$$

Item (82) implies that we have a derivation of

$$\Gamma'\{f^* : \phi_1 \xrightarrow{\Delta}_* \phi_2\}\{x^* : \phi_1\} \rhd e' : (\phi_2, \Delta'') \Rightarrow m' \tag{86}$$

in which $\Delta'' \subseteq \Delta \uplus \{x^*, f^*\}$, $f^* \notin \mathsf{LV}(\phi_1)$ and $x^*, f^* \notin \mathsf{LV}(\phi_2)$. Using items (85) and the property $f^* \notin \mathsf{LV}(\phi_1)$ we can apply Lemma 9.4 to obtain

$$\mathsf{Consistent}(\pi, v_2, w_2, \phi_1, \Delta_a \uplus \{f^*\}). \tag{87}$$

Using $(\mathsf{LV}(\phi_1) \cap (\Delta_a \uplus \{f^*\})) \subseteq \mathsf{dom}(\Gamma'\{f^* : \phi_1 \xrightarrow{\Delta}_* \phi_2\})$ (which follows from item (84)) and items (83) and (87), we can apply rule (6) of Definition 5.3 to obtain

$$\begin{aligned}\mathsf{Consistent}(&\rho'\{f \mapsto [\rho', \mu f.\lambda x.e']\}\{x \mapsto v_2\}, \pi\{x^* \mapsto w_2\},\\ &\eta', \Gamma'\{f^* : \phi_1 \xrightarrow{\Delta}_* \phi_2\}\{x^* : \phi_1\}, \Delta_a \uplus \{x^*, f^*\}).\end{aligned} \tag{88}$$

Observe that $\Delta'' \subseteq (\Delta \uplus \{x^*, f^*\}) \subseteq (\Delta_a \uplus \{x^*, f^*\})$. Using this property, items (50), (86) and (88), we can apply the inductive hypothesis to Π_3, yielding that there exists some w and Θ_3 such that

$$\Theta_3 :: \pi\{x^* \mapsto w_2\}; \eta' \rhd m' \hookrightarrow_t w, \tag{89}$$

$$\mathsf{Consistent}(\pi\{x^* \mapsto w_2\}, v, w, \phi_2, \Delta_a \uplus \{x^*, f^*\}). \tag{90}$$

By Proposition 9.5, item (90) and $x^* \notin \mathsf{LV}(\phi_2)$ imply

$$\mathsf{Consistent}(\pi, v, w, \phi_2, \Delta_a \uplus \{f^*\}). \tag{91}$$

By Proposition 9.4, item (91) and $f^* \notin \mathsf{LV}(\phi_2)$ imply

$$\mathsf{Consistent}(\pi, v, w, \phi_2, \Delta_a). \tag{92}$$

With some simple reasoning we can show that $\Delta_a \cap \mathsf{LV}(\phi_2) = \Delta' \cap \mathsf{LV}(\phi_2)$. From this and item (92), Proposition 9.4 yields

$$\mathsf{Consistent}(\pi, v, w, \phi_2, \Delta'). \tag{93}$$

Finally we can construct the deduction Θ as

$$\frac{\overset{\Theta_1}{\pi; \eta \rhd m_1 \hookrightarrow_t [\eta', \lambda x^*.m']} \qquad \overset{\Theta_2}{\pi; \eta \rhd m_2 \hookrightarrow_t w_2} \qquad \overset{\Theta_3}{\pi\{x^* \mapsto w_2\}; \eta' \rhd m' \hookrightarrow_t w}}{\pi; \eta \rhd (m_1 \mathbin{@^*} m_2) \hookrightarrow_t w}.$$

$\square$

References

Amtoft, T. (1993) Minimal thunkification. In: Cousot, P., Falaschi, M., Filè, G. and Rauzy, A., editors, *Proceedings of the 3rd International Workshop on Static Analysis: Lecture Notes in Computer Science 724*, pp. 218–229. Springer-Verlag.

Appel, A. W. (1992) *Compiling with Continuations.* Cambridge University Press.

Baker-Finch, C. A. (1994) Type theory and projections for higher-order static analysis. In: Sestoft, P. and Søndergaard, H., editors, *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp 43–52. ACM SIGPLAN, ACM Press.

Banerjee, A. and Schmidt, D. A. (1994) Stackability in the simply-typed call-by-value lambda calculus. In: Charlier, B. L., editor, *First International Static Analysis Symposium: Lecture Notes in Computer Science 864*, pp. 131–146. Springer-Verlag.

Barendregt, H., (1984) *The Lambda Calculus: Its Syntax and Semantics: Studies in Logic and the Foundations of Mathematics 103.* North-Holland.

Burns, P. H. (1996) A space efficient implementation of a functional language. *Honors thesis*, Pennsylvania State University, University Park, PA.

Damas, L. (1985) Type Assignment in Programming Languages. *PhD thesis* University of Edinburgh. Available as CST-33-85.

Goldberg, B. and Park, Y. G. (1990) Higher order escape analysis: Optimizing stack allocation in functional program implementations. In: Jones, N., editor, *Proceedings of the 3rd European Symposium on Programming: Lecture Notes in Computer Science 432*, pp. 152–160. Springer-Verlag.

Hannan, J. and Miller, D. (1992) From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, **2**(4): 415–459. (Appears in a special issue devoted to the 1990 ACM Conference on Lisp and Functional Programming.)

Hannan, J. and Pfenning, F. (1992) Compiler verification in LF. In: Scedrov, A., editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pp. 407–418. IEEE Press.

Hannan, J. (1995) Type systems for closure conversions. In: Nielson, H. R. and Solberg, K. L., editors, *Participants' Proceedings of the Workshop on Types for Program Analysis*, pp. 48–62. Aarhus University, DAIMI PB-493.

Ibarra, M. E. (1997) Verification of a type-based escape analysis for functional languages in a logical framework. *Master's thesis*, Pennsylvania State University, University Park, PA.

Kranz, D. (1988) ORBIT: An Optimizing Compiler for Scheme. *PhD thesis*, Yale University.

Kuo, T.-M. and Mishra, P. (1989) Strictness analysis: A new perspective based on type inference. *Fourth International Conference on Functional Programming and Computer Architecture*, pp. 260–272. ACM Press.

Milner, R. and Tofte, M. (1991) Co-induction in relational semantics. *Theoretical Computer Science*, **87**(1): 209–220.

Milner, R. (1978) A theory of type polymorphism in programming. *J. Computer and System Sciences*, **17**(3): 348–375.

Pfenning, F. (1991) Logic programming in the LF logical framework. In: Huet, G. and Plotkin, G., editors, *Logical Frameworks*, pp. 149–181. Cambridge University Press.

Sestoft, P. (1991) Analysis and Efficient Implementation of Functional Programs. *Rapport nr 92/6*, DIKU, Copenhagen.

Shao, Z. and Appel, A. W. (1994) Space-efficient closure representations. *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 150–161. ACM Press.

Tofte, M. and Talpin, J.-P. (1994) Implementation of the typed call-by-value λ-calculus using a stack of regions. *Conf. Rec. 21st ACM Symposium on Principles of Programming Languages*, pp. 188–201.

Wand, M. and Steckler, P. (1997) Lightweight closure conversion. *ACM Trans. Program. Lang. Syst.*, **19**(1): 48–86.

Wright, D. A. (1991) A new technique for strictness analysis. In: Abramsky, S. and Maibaum, T., editors, *TAPSOFT '91*, pp. 235–258.