Electronic Notes in Theoretical Computer Science 45 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume45.html 12 pages

# On Extracting Static Semantics

John Hannan

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

**Abstract**

We examine the problem of automatically extracting a static semantics from a language's semantic definition. Traditional approaches require manual construction of static and dynamic semantics, followed by a proof that the two are consistent. As languages become more complex, the static analyses also become more complex, and consistency proofs have typically been challenging. We need to find techniques for automatically constructing static analyses that are provably correct.

## 1 Introduction

A common approach to programming language design and implementation is to identify static (compile time) and dynamic (run time) operations or properties of the language and then construct a static semantics and a dynamic semantics. Even if formal language definitions do not consist of such explicit phases, compilers for these languages almost always do make such distinctions, performing semantic analysis before generating code.

A simple compiler-design principle might be summarized by the following phases:

(i) Lexical Analysis
(ii) Semantics Analysis
(iii) Code Generation

Alternatively, the approach to language design mentioned above can be summarized by:

(i) Static Semantics
(ii) Dynamic Semantics

[1] This work is supported in part by NSF Award #CCR-9900918.
[2] Email: hannan@cse.psu.edu

These two phases correspond to the latter two phases of a compiler. Just as code generation uses information provided by the semantics analysis, the dynamic semantics can use information or properties provided by the static semantics. In a strongly typed language, this information consists of a well-typed result that allows the dynamic semantics to avoid any typechecking.

Because types play a significant role in most static analyses, the static phase of a language's definition is typically given as a type system. Types characterize the meaning of expressions. To ensure that that the static semantics provides valid information to the dynamic semantics, some kind of consistency result is required. This result states that an expression's value will have the same type as the expression, and hence, type errors will not occur at runtime.

More generally, we can imagine describing program properties other than types. These might include properties of security, resource usage, or effect. Our goal for describing these properties might be to provide compile-time analyses that ensure certain runtime behaviors of programs satisfying these properties. For example, we might describe a property about security involving levels of priority and access to secure information. A static analysis that succeeds should tell us that at run time, no security violations can occur. Again, we must provide a consistency result that demonstrates the correctness of the static analysis: assertions made by the static analysis are verified/reflected in the dynamic semantics. As the properties become more complex and the analyses become more involved, proving this kind of consistency is increasingly difficult.

Separating semantics and compilers into two phases provides two advantages: efficiency and predictability. By performing tests and operations once at compile time, we may avoid performing them at all during run time. Static type checking of strongly typed languages is a good example of this. By establishing properties of a program at compile time, we have some prediction or guarantee of the behavior of the program at run time. Again, static type checking may ensure that a program will not generate any run-time type errors. Generally, these advantages outweigh the disadvantages of having to construct two distinct phases and then prove them correct.

However, as we define languages with more complex features and attempt to provide more information through static analyses, the difficulty of defining these analyses and proving them consistent with the dynamic semantics will grow. An alternative is to define a single semantics that contains both the static and dynamic checks. This is typically easier than defining two semantics, in part because checking of properties can be done when the properties are clearly evident. For example, instead of providing static typechecking, we use dynamic type checking. Then we might only check that values have the appropriate type. Of course, this might result in a different semantics.

One way of viewing such a combined semantics is that it manipulates both *syntactic objects* (available at compile time) and *semantic objects* (available at

run time). Starting with such a semantics we would like to consider the task of separating this semantics into two parts: a traditional static and dynamic semantics such that we are guaranteed consistency between the two.

## 2 Staging Semantics

The idea of constructing a type checker from a semantic specification is not new, though little work has been done in this area. In 1991, Neil Jones considered this problem [4]. His focus is specifically on type systems, not the more general problem of static properties. But as types are the most important static property in use today, starting with them is sensible.

Since the problem of constructing a type checker from a semantics is one of introducing stages of computation - compile time and run time - it's only natural to consider traditional staging transformations [5], particularly partial evaluation, as a technique for separating the static and dynamic parts of a semantics. To do this, Jones reviews some standard technology for discussing partial evaluation.

First we should understand exactly what the problem is. Given an operational semantics that manipulates both syntactic objects and semantic objects, we want to (automatically) separate compile-time operations and runtime operations. We are specifically concerned with error checking.

Let $\mathbf{L}$ be a programming language. Then $[\![p]\!]_{\mathbf{L}} v$ is the result of running $\mathbf{L}$-program $p$ on input $v$. A *Definitional interpreter int* for language $\mathbf{S}$ written in $\mathbf{L}$ has the following property:

$$[\![int]\!]_{\mathbf{L}}(source, input) \;=\; [\![source]\!]_{\mathbf{S}}\, input$$

Errors in $\mathbf{S}$-programs are realized by error-detecting code in *int*. Given this setting we are able to describe static semantics:

> *Static Semantics* is a mechanism for detecting whether or not a given $\mathbf{S}$-program can cause *int* to execute a "static error" on some input.

The notion of static error is interesting. Do we take static semantics as providing the definition of what the static errors are? Or is there some other definition of static error from which our definition of static semantics can be proved sound and complete? Considering this further, we see that we really have to address two problems in general:

(i) Determine the static properties of a semantics

(ii) Construct a static semantics that captures these properties

The question of whether static semantics define the static errors is analogous to existing views on type systems:

(i) *Prescriptive* (Church): types are predefined conditions to ensure meaningfulness;

(ii) *Descriptive* (Curry): types describe the values that a program manipu-

lates.

We do not concern ourselves with the differences of these viewpoints. Instead, we will consider a semantics that contains a descriptive use of types and attempt to construct a semantics that uses the prescriptive view. In particular, we would like to construct a typechecker from a given semantics. More generally, we would like the typechecking phase to occur before the evaluation phase in the language's definition (semantics).

Following Jones [4], we can make the concept of static error more precise, using the concept of program specialization called partial evaluation. Assume we have a partial evaluator *mix*. Recall that a partial evaluator takes a program $p$ and part of the program's input $s$ (the static part) and yields a new program that when applied to the dynamic part $d$ behaves as the original program does:

$$mix(p, s)\,(d) \;=\; p(s, d)$$

Partial evaluation provides a staging of computation into two parts, typically described as static part (the computation $mix(p, s)$) and the dynamic part (the application of the residual program to $d$).

Of interest to us in when the program is a definitional interpreter. In this case the static input is the source program to be interpreted, (the dynamic input is the input to the source program) and the above equation can be rewritten as

$$int_{source} \;=\; [\![mix]\!]_{\mathbf{L}}(int, source)$$

If the interpreter contains error checking (including typechecking) then we might hope that partial evaluation performs these tests and that the residual program $p_s = mix(p, s)$ contains no such error checking. In this case, we have solved the problem of staging typechecking, though without constructing an explicit type system. Unfortunately, this idea doesn't work in practice because the restriction of no error checking in the residual program is too strong, as this rules out any kind of errors, even dynamic ones (e.g., divide by zero), from ever occurring in programs.

The problem that arises with this approach is that we fail to distinguish between static and dynamic errors. A further problem is that this approach deals with programs one at a time. We do not have any guarantee that every source program processed in this way will result in a residual program that contains no error checking. We have no way of determining if our language is strongly typed (meaning here that all static-error testing can be performed by partial evaluation).

To solve the problem of distinguishing between static and dynamic errors Jones observes that a static error should be one that can only be performed by the partial evaluator (operating on *int*) and code for it will never be generated in a residual program. Such a property can be detected by a binding-time analysis (BTA). BTA divides every basic function call or data constructor into one of two classes: those that can be computed at program specialization time,

and those that must be performed at run time (by a specialized program $p_s$) [4]. Thus the problem of identifying static errors in an interpreter reduces to performing a binding-time analysis on it when we consider the source program as static input.

Jones demonstrates how BTA, combined with partial evaluation, can be used to identify static properties (type checking) and perform them separately from the evaluation of the program.

## 3 An Alternative Approach

A fundamental limitation of using partial evaluation to specify static semantics is that we do not explicitly construct a type system or some other specification of the static operations. Instead, a general partial evaluator implicitly performs these operations. An alternative to partial evaluation is another staging transformation called pass separation [5]. Recall that staging transformations are, in general, an methods of separating stages of computations based on the availability of data, with the most common application being developing compilers from interpreters. Partial evaluation is perhaps the most widely known and used staging transformation, but others exist, including traditional compiler optimizations (e.g., constant folding) and pass separation.

Consider the general problem addressed by staging transformations: Given a program $p$ with two inputs $x, y$, *stage* the computations performed by $p(x, y)$ into two sequential phases:

- one taking $x$ as input
- one taking $y$ as input

Partial evaluation accomplishes this via a single program *mix* such that if $mix(p, x) = p_x$ then $p(x, y) = p_x(y)$ for all inputs $x$ and $y$. In the example from above, $p$ was an interpreter, $x$ the source program to be interpreted, and $y$ the input to the program. Pass separation requires a technique to construct two new programs, $p_1$ and $p_2$ such that $p(x, y) = p_2(p_1(x), y)$ for all inputs $x$ and $y$. In this case, if $p$ is an interpreter and $x$ a source program, then we might expect $p_1$ to be a compiler and $p_2$ to be the evaluator for the target language of the compiler. The drawback of pass separation is that we do not have a general way of taking an arbitrary program $p$ and constructing the required new programs $p_1$ and $p_2$ such that $p_1$ performs some non-trivial computations. (We could always let $p_1$ be the identity function and let $p_2 = p$.) Arbitrary functions are just too general, and the possible divisions into two functions too numerous to support a simple means for performing pass separation.

In previous work we addressed this problem by considering a restricted form of interpreter, namely abstract machines [2]. In this work, an abstract machine is represented by a set of rewrite rules $s \Rightarrow s'$ in which $s$ and $s'$ are machine states. Typically, machine states consist of program and data. Each rule of the machine specifies how a given program state manipulates data to

$$\langle (e_1\, e_2)\, ;\, C,\ \ (\rho\, ;\, L,\ S)\rangle \Rightarrow \langle e_1\, ;\, e_2\, ;\, \mathsf{ap}\, ;\, C,\ \ (\rho\, ;\, \rho\, ;\, L,\ S)\rangle$$

$$\langle (\lambda\, e)\, ;\, C,\ \ (\rho\, ;\, L,\ S)\rangle \Rightarrow \langle C,\ \ (L,\ \{\rho, \lambda\, e\}\, ;\, S)\rangle$$

$$\langle 1\, ;\, C,\ \ ((\rho; v)\, ;\, L,\ S)\rangle \Rightarrow \langle C,\ \ (L,\ v\, ;\, S)\rangle$$

$$\langle (e+1)\, ;\, C,\ \ ((\rho; v)\, ;\, L,\ S)\rangle \Rightarrow \langle e\, ;\, C,\ \ (\rho\, ;\, L,\ S)\rangle$$

$$\langle \mathsf{ap}\, ;\, C,\ \ (L,\ v\, ;\, \{\rho, \lambda\, e\}\, ;\, S)\rangle \Rightarrow \langle e\, ;\, C,\ \ ((\rho; v)\, ;\, L,\ S)\rangle$$

Fig. 1. The CLS Machine

yield a new program state. Rewriting in this system is particularly simple as we only allow rewriting at the top level, i.e., the entire term. This simple form a computation is expressive, but also extremely convenient to reason about because we only have a single form of computation (machine-state rewrite) to consider.

As an example we studied the CLS machine [3], a variant of Landin's SECD machine [6]. This is a machine that performs call-by-value reduction on the lambda calculus. The machine state consists of a triple $\langle C, L, S\rangle$ consisting of

*Code* - a sequence of expressions to be evaluated

$L$ - a sequence of environments

$S$ - a stack of intermediate values

The terms of the language are given by the following grammar:

$$e ::= \ 1 \mid e+1 \mid \lambda\, e \mid e\, e \mid \mathsf{ap}$$

Code or a program is given by a list of terms. An environment is a list of values. Values are simply closures consisting of a term and an environment. A stack is also a list of values. The rules for the machine are given in Figure 1

The idea behind our pass-separation technique is to consider abstract machines with rules of the form

$$\langle s, d\rangle \longrightarrow \langle s', d'\rangle$$

in which $s$ and $s'$ represent static (compile-time) components and $d$ and $d'$ represent dynamic (runtime) components. For the CLS machine we take the $C$ component to be the static part and the pair $(L, S)$ to be the dynamic part. We then decompose each rule into two parts: one operating only on the static part and one operating on the dynamic part. For example, a rule of the form

$$\langle s, d\rangle \longrightarrow \langle s', d'\rangle$$

would be decomposed into a pair of rules

$$\langle s, Y\rangle \longrightarrow \langle s_1, Y\rangle \ \text{ and } \ \langle s_2, d\rangle \longrightarrow \langle s', d'\rangle.$$

The terms $s_1$ and $s_2$ are constructed to ensure that rewriting proceeds in lock

$$\mathsf{ev}(e_1\ e_2)\ ;\ C \Rightarrow \mathsf{push}\ ;\ \mathsf{ev}(e_1)\ ;\ \mathsf{ev}(e_2)\ ;\ \mathsf{ap}\ ;\ C$$
$$\mathsf{ev}(\lambda\, e)\ ;\ C \Rightarrow \mathsf{lam}(\mathsf{ev}(e)\ ;\ \mathsf{nil})\ ;\ C$$
$$\mathsf{ev}(1)\ ;\ C \Rightarrow \mathsf{fst}\ ;\ C$$
$$\mathsf{ev}(e+1)\ ;\ C \Rightarrow \mathsf{snd}\ ;\ \mathsf{ev}(e)\ ;\ C$$

$$\begin{array}{rcl}
\langle \mathsf{push}; C,\ \rho\ ;\ L,\ S\rangle & \Rightarrow & \langle C,\ \rho\ ;\ \rho\ ;\ L,\ S\rangle \\
\langle (\mathsf{lam}\, C'); C,\ \rho\ ;\ L,\ S\rangle & \Rightarrow & \langle C,\ L,\ (\rho, \mathsf{lam}\, C')\ ;\ S\rangle \\
\langle \mathsf{fst}; C,\ (v,\rho)\ ;\ L,\ S\rangle & \Rightarrow & \langle C,\ L,\ v\ ;\ S\rangle \\
\langle \mathsf{snd}; C,\ (v,\rho)\ ;\ L,\ S\rangle & \Rightarrow & \langle C,\ \rho\ ;\ L,\ S\rangle \\
\langle \mathsf{ap}; C,\ L,\ v\ ;\ (\rho, \mathsf{lam}\, C')\ ;\ S\rangle & \Rightarrow & \langle C'C,\ (v,\rho)\ ;\ L,\ S\rangle
\end{array}$$

Fig. 2. The Separated Machines

step with the original system. Applying this simple transformation yields two sets of rewrite rules (abstract machines):

- a compiler (introducing a new machine language)
- an interpreter for the new language

The construction of these new rules is based on the form of the original rules. Because the rules have such a simple structure, we can identify a few distinct cases that give rise to meta-rules - rules on how to construct new rules. These meta-rules are based on the dependencies among $s$, $d$, $s'$, and $d'$. In particular, we consider how the terms $s'$ and $d'$ are constructed from $s$ and $d$.

Applying these meta-rules to the CLS machine we arrive at the two machines given in Figure 2. The new machine language consists of instructions push, lam, fst, and snd. These instructions are generated by the meta-rules and the rules giving meaning to them provide a definition for this new abstract machine language.

The resulting machines can be proved correct with respect to the original machine. Interestingly, this new machine is essentially the Categorical Abstract Machine [1]. The compiler decomposes the structure of a lambda term into simpler components (new machine language), so we see a separation between the traversal of the original term's structure and the actual evaluation of the term.

## 4 Pass Separation for Static Semantics

Can pass separation be used to separate the static semantics inherent in an abstract machine? We explore this idea by considering the issues involved and give an example constructed through some reverse engineering. Three questions immediately must be considered:

- Can we identify an appropriate form of the operational semantics? Even

restricting ourselves to abstract machines, we still want want to understand the best form that will allow us to separate out a static semantics.

- Can we identify static errors? A more general problem is simply identifying the static properties of a semantics. This is analogous to defining what static errors are.
- Can we identify a set of meta-rules for decomposing static and dynamic properties? Our previous work using pass separation identified a small set of meta-rules for constructing new rewrite rules. These meta-rules were justified with correctness proofs. We would like to find a general set for the current problem

We do not give any definitive answers to these questions. Instead we will work through an example to demonstrate that the ideas of pass separation can be applied to the problem of separating static semantics.

We begin by modifying the abstract machine providing the semantics for our language. The CLS machine is an untyped machine: no typechecking is done in the machine; terms have no associated types. To provide the most information possible, we initially assume that terms are explicitly typed: every term (and subterm) is explicitly tagged with its type (written as a superscript). While this is perhaps not realistic (as it assumes some pre-existing type system that inserts these types), this assumption is a useful starting point allowing us to see how we might separate static and dynamic operations. We modify the CLS machine to manipulate such explicitly typed terms and to include certain type checks. In particular, the machine checks that the value of an argument in a function call has the same type as the formal parameter bound to it. The machine also checks that the value of a variable (found in an environment) has the same type as the variable. The modified machine is given in Figure 3. An environment $\delta$ maps variables (de Bruijn indices) to typed values (i.e., values and their types).

We can restructure this machine by making some simple syntactic changes. Specifically, we decompose $\delta$ into a pair of environments: $\Gamma$ (mapping variables to types) and $\rho$ (mapping variables to values). The resulting machine is given in Figure 4. We take as given that the static property we want to extract is the association of an expression and a type. Automatically identifying this property and structuring the machine in such a way as to make this property evident is a challenging problem, and one for which we currently offer no solution. But once we have the machine in Figure 4, we can consider how to achieve our goal. Ultimately, our goal is not simply to produce a static semantics, but to provide some static checking that when satisfies indicates an absence of certain kinds of runtime errors.

Because our goal is to avoid error states in the abstract machine we will attempt to identify states that can never lead to an error state. Fortunately, we have explicit descriptions of error states in our machine. How do we know if a particular machine state can ever lead to an error state? Forward reasoning

$$\langle (m\,n)^{\tau}\,;C,\ \ (\delta\,;L,\ S)\rangle \Rightarrow \langle m^{\tau_2\to\tau}\,;n^{\tau_2}\,;\mathsf{ap}\,;C,\ \ (\delta\,;\delta\,;L,\ S)\rangle$$

$$\langle (\lambda\,m)^{\tau_1\to\tau_2}\,;C,\ \ (\delta\,;L,\ S)\rangle \Rightarrow \langle C,\ \ (L,\ \{\delta,\lambda\,m\}^{\tau_1\to\tau_2}\,;S)\rangle$$

$$\langle 1^{\tau}\,;C,\ \ ((\delta;v^{\tau})\,;L,\ S)\rangle \Rightarrow \langle C,\ \ (L,\ v\,;S)\rangle$$

$$\langle 1^{\tau}\,;C,\ \ ((\delta;v^{\tau'})\,;L,\ S)\rangle \Rightarrow \mathit{error}\ \ \mathsf{if}\ \tau\neq\tau'$$

$$\langle (m+1)^{\tau}\,;C,\ \ ((\delta;v)\,;L,\ S)\rangle \Rightarrow \langle m^{\tau}\,;C,\ \ (\delta\,;L,\ S)\rangle$$

$$\langle \mathsf{ap}\,;C,\ \ (L,\ v^{\tau_2}\,;\{\delta,\lambda\,m\}^{\tau_2\to\tau}\,;S)\rangle \Rightarrow \langle m^{\tau}\,;C,\ \ ((\delta;v^{\tau_2})\,;L,\ S)\rangle$$

$$\langle \mathsf{ap}\,;C,\ \ (L,\ v\,;\{\delta,\lambda\,m\}\,;S)\rangle \Rightarrow \mathit{error}\ \ \mathsf{otherwise}$$

Fig. 3. Explicitly Typed Machine

$$\langle [\Gamma,(m\,n)]^{\tau}\,;C,\ \ \rho\,;L,\ S\rangle \Rightarrow \langle [\Gamma,m]^{\tau_2\to\tau}\,;[\Gamma,n]^{\tau_2}\,;\mathsf{ap}\,;C,\ \ \rho\,;\rho\,;L,\ S\rangle$$

$$\langle [\Gamma,\lambda\,m]^{\tau_1\to\tau_2}\,;C,\ \ \rho\,;L,\ S\rangle \Rightarrow \langle C,\ \ L,\ \{\rho,[\Gamma,\lambda\,m]^{\tau_1\to\tau_2}\}\,;S\rangle$$

$$\langle [(\Gamma;\tau),1]^{\tau}\,;C,\ \ (\rho;v)\ L,\ S\rangle \Rightarrow \langle C,\ \ L,\ v\,;S\rangle$$

$$\langle [(\Gamma;\tau'),1]^{\tau}\,;C,\ \ (\rho;v)\ L,\ S\rangle \Rightarrow \mathit{error}\ \ \mathsf{if}\ \tau\neq\tau'$$

$$\langle [(\Gamma;\tau'),(m+1)]^{\tau}\,;C,\ \ (\rho;v)\ L,\ S\rangle \Rightarrow \langle [\Gamma,m]^{\tau}\,;C,\ \ \rho\ L,\ S\rangle$$

$$\langle \mathsf{ap}\,;C,\ \ L,\ v^{\tau_2}\,;\{\rho,[\Gamma,\lambda\,m]^{\tau_2\to\tau}\}\,;S\rangle \Rightarrow \langle [(\Gamma;\tau_2),m]^{\tau}\,;C,\ \ (\rho;v)\ L,\ S\rangle$$

$$\langle \mathsf{ap}\,;C,\ \ L,\ v\,;\{\rho,\Gamma,\lambda\,m\}\,;S\rangle \Rightarrow \mathit{error}\ \ \mathsf{otherwise}$$

Fig. 4. A Restructured Machine

does not seem to help much here, but backward reasoning does. If a machine is in a safe state, then the previous state it was in led to that safe state. We can give an inductive definition of safe state:

(i) Any final state is a safe state;

(ii) if $s$ is a safe state and $s' \Rightarrow s$ is an instance of some rule in the machine then $s'$ is also a safe state.

Note that condition ii requires that the machine be deterministic: the only step possible from state $s'$ must be to state $s$. This kind of condition is part of the formal theory that must be developed to formally justify the steps we

$$\frac{[\Gamma, m]^{\tau_2 \to \tau} \qquad [\Gamma, n]^{\tau_2}}{[\Gamma, (m\, n)]^{\tau}}$$

$$\frac{[\Gamma, \lambda\, m]^{\tau_1 \to \tau_2}}{[\Gamma, \lambda\, m]^{\tau_1 \to \tau_2}}$$

$$\frac{}{[(\Gamma; \tau), 1]^{\tau}}$$

$$\frac{[\Gamma, m]^{\tau}}{[(\Gamma; \tau'), (m+1)]^{\tau}}$$

$$\frac{[(\Gamma; \tau_2), m]^{\tau}}{[\Gamma, \lambda\, m]^{\tau_1 \to \tau_2}}$$

Fig. 5. Inference Rules for Static Typechecking

are taking to construct a static semantics.

Now that we have a general strategy for determining safe states we would like to extract the static properties. Since the type errors we wish to avoid are based on the property of type associated with expressions, we can understand error states to be states in which an incorrect type is associated with an expression. This observation leads us to a simple meta-rule for constructing inference rules for a static semantics:

> For each rewrite rule $s \Rightarrow s'$ in the machine, let $A$ and $A'$ be the set of static properties occurring in $s$ and $s'$, respectively. If $A$ is not a singleton set, then the method fails; otherwise construct the inference rule
>
> $$\frac{A'}{A}$$

If we apply this meta-rule to the rules in Figure 4, then we get the set of inference rules in Figure 5. The second rule provides no information so it can be deleted. The remaining rules are the traditional rules for typechecking the lambda calculus.

While we have informally justified the construction of these rules, arguing that the original rewrite rules provide enough information to extract inference rules, we have not provided a formal proof that the inference system provides safety against runtime type errors from occurring. Neither have we charac-

terized a general form of abstract machines for which this technique applies. Still, we hope this example sheds some light on the problem of extracting a static semantics from an operational one.

## 5 Further Work and Conclusions

Where did the types in the previous section come from? We assumed that terms came with their types already attached and that the semantics used those types to perform checks. These assumptions stacked the deck in our favor, helping us to achieve our goal of constructing a type system. As a first step, we think this is fair, providing us with as much information as possible. One might argue that what our example really shows is part of the proof of type soundness, telling us that we can erase the types at runtime. But where did these types come from in the first place?

A more realistic starting point for our work would be the original CLS machine. It contains no types. Error conditions are implicit: if a machine state does not match the left-hand side of some rule, then an error occurs (unless the machine is in a final state). A more familiar starting point may be a language like Scheme. Scheme has no static typing of expressions but it does have dynamic typechecking on values. For example, given an application $(e_1\, e_2)$ Scheme only requires that the value of $e_1$ be a function (of one argument). There is no check that the type of the value of $e_2$ match the type of the parameter of this function.

Suppose our goal is to start with a language like Scheme but then add static checking to ensure that certain kinds of runtime errors cannot occur. Essentially, how can we get from a language like Scheme (with dynamic typing of values) to one like ML (with static typing of expressions)? We argue that a technique similar to the one proposed in the previous section may work. We start with the very simple notion of type that Scheme employs. We then identify a notion of safe state. This will be more complicated than in the previous example. Again, we must work backwards using the rewrite rules, arguing that safe states come from safe states. We believe that this approach is worth studying for it may lead to some interesting results.

In summary, the study of type systems and static analyses done by Neil Jones provided a starting point for studying the problem of extracting static semantics. The identification of key problems, including definitions of static errors, helped pave the way for the current and future work.

## References

[1] Cousineau, G., P.-L. Curien and M. Mauny, *The categorical abstract machine*, The Science of Programming **8** (1987), pp. 173–202.

[2] Hannan, J., *Staging transformations for abstract machines*, in: P. Hudak and

N. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation* (1991), pp. 130–141.

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

[4] Jones, N., *Static semantics, types, and binding time analysis*, Theoretical Computer Science **90** (1991), pp. 95–118, also appeared in Images of Programming, eds. D. Bjørner and V. Kotov, North-Holland.

[5] Jørring, U. and W. Scherlis, *Compilers and staging transformations*, in: *Thirteenth ACM Symposium on Principles of Programming Languages*, 1986, pp. 86–96.

[6] Landin, P. J., *The mechanical evaluation of expressions*, Computer Journal **6** (1964), pp. 308–320.