Model Verification in $\lambda\sigma$: A Type Inference Approach

Enrique V. Kortright

Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504

Abstract

In this paper we describe a number of model analysis and verification operations based on type inference in the $\lambda\sigma$ simulation language. $\lambda\sigma$ is a simulation language based on the typed $\lambda$-calculus. $\lambda\sigma$ entities correspond to typed $\lambda$-expressions, while $\lambda\sigma$ activities correspond to subtypes. Thus, entities can be generated by means of type-introduction rules, and operations can be defined on entities by means of type elimination and equality rules. Premises of the form $e \in \tau$ in an introduction rule used to create a new entity can be satisfied by substituting for $e$ any entity of type $\tau$ in a neighboring activity. It is then possible to perform a number of model analysis and verification operations using type inference algorithms available for the typed $\lambda$-calculus.

1 Introduction

$\lambda\sigma$ is a new simulation language based on the typed $\lambda$-calculus [4,5]. $\lambda\sigma$ entities correspond to typed $\lambda$-expressions, while $\lambda\sigma$ activities correspond to subtypes. Thus, entities can be generated by means of type-introduction rules, and operations can be defined on entities by means of type elimination and equality rules [7]. Premises of the form $e \in \tau$ in an introduction rule used to create a new entity can be satisfied by substituting for $e$ any entity of type $\tau$ in a neighboring activity.

It is possible to perform a number of model analysis and verification operations using type inference algorithms available for the typed $\lambda$-calculus. A type inference algorithm determines the type of an expression from the types of its component subexpressions and from assumptions about the types of its atomic subexpressions (see [8]). It is then possible to use a type inference algorithm to perform the following operations:

- Verify (type check) all expressions in a model. This includes the verification of all expressions of the form $e \in \tau$ (that is, $e$ is of type $\tau$), and all function applications (that is, the application of an operation to an entity).
- Given an activity, verify that all premises in the introduction rules that it uses to create its entities can be satisfied by the entities of at least one neighboring activity.
- Given an activity, verify that the entities generated in it can potentially be used to satisfy a premise in an introduction rule used by a neighboring activity, thus potentially flowing through the activity network.
- Given an entity, determine the activities in the model which can possibly generate it.

In this way, it is possible to guarantee that a simulation will run free of errors caused by type incompatibilities. More importantly, the verifications proposed in this paper can increase the validity of a model. Among other errors, these methods detect:

- the error of specifying the creation of an entity without providing a path for the entities required by the introduction rule used, and
- the error of defining a path for an entity to exit an activity without providing for its use in the target activity, that is, without the existence of a premise requiring such an entity in some introduction rule.

2 The typed $\lambda$-calculus

The $\lambda$-calculus was proposed by Church for the study of the formal properties of functions as defined by rules,
rather than in set theoretic terms [1]. The lambda calculus is a type-free theory in which any object can be used as a function or as an argument to a function.

A type system for the lambda calculus consists of a set of types together with rules for assigning a type to lambda calculus expressions. A type system for the lambda calculus can be described by a set of Gentzen-style (natural deduction) inference rules. Such rules can be classified as follows [7]. Type formation rules are used to determine what counts as a type. Type introduction rules are used to describe how the elements of a type can be constructed and how it can be decided whether two elements of the same type are equal. We can define new functions on types by means of elimination rules and define how such functions operate on elements of the appropriate types by means of equality rules. From the type introduction rules, it is possible to derive type inference rules describing how a type can be assigned to a lambda-calculus expression. If a type can be assigned to a lambda-expression using the type inference rules, then the expression is well-typed.

Assume that C and V are a set of constants and variables, respectively. We define the set of lambda-expressions, denoted by lambda-exp, to be the smallest set closed under the following rules:

**Constant-expressions**

\[
\frac{c \in C}{c \in \lambda\text{-exp}}
\]

**Variable-expressions**

\[
\frac{x \in V}{x \in \lambda\text{-exp}}
\]

**lambda-abstraction**

\[
\frac{x \in \text{variable} \quad e \in \lambda\text{-exp}}{\lambda x. e \in \lambda\text{-exp}}
\]

**Application**

\[
\frac{e_1, e_2 \in \lambda\text{-exp}}{e_1 e_2 \in \lambda\text{-exp}}
\]

**Tuples**

\[
\frac{e_1, \ldots, e_m \in \lambda\text{-exp}}{\langle e_1, \ldots, e_m \rangle \in \lambda\text{-exp}}
\]

**Projection-functions**

\[
\frac{e \in \lambda\text{-exp}}{e[i] \in \lambda\text{-exp}}, \text{ for every integer } i
\]

**Selective-modifications**

\[
\frac{e_1, e_2 \in \lambda\text{-exp}}{e_1([i] = e_2) \in \lambda\text{-exp}}
\]

We define the type system T as the smallest set that satisfies the following type rules (see [7] for a more complete version of this type system):

**Basic-types**

\[
a \in T,
\]

for any user-specified basic type a.

**x-formation**

\[
A, B \in T \quad \Rightarrow \quad A \times B \in T
\]

**x-introduction**

\[
a \in A \quad b \in B \quad a = c \in A \quad b = d \in B \quad \Rightarrow \quad [a, b] = [c, d] \in A \times B
\]

**x-elimination**

\[
\frac{c \in A \times B \quad c \in A \times B}{c[1] \in A \quad c[2] \in B}
\]

**x-equality**

\[
\frac{c \in A \times B}{[c[1], c[2]] = c \in A \times B}
\]

**+ -formation**

\[
A, B \in T \quad \Rightarrow \quad A + B \in T
\]

**+ -introduction**

\[
a \in A \quad b \in B \quad A a.b \in A + B
\]

**+ -elimination**

\[
\frac{c \in A + B \quad a \in A}{c[a] \in B}
\]

**+ -equality**

\[
\frac{c \in A + B \quad a \in A}{\lambda a.ca \equiv c \in A \rightarrow B}
\]

In Section 4, we describe a set of inference rules that will assign a type to every lambda-expression.
3 The $\lambda \sigma$ modeling and simulation language

The typed $\lambda$-calculus has been used to formalize intuitionistic logic [7] and programming languages [8,3]. In order to formalize intuitionistic logic, Martin-Löf [7] identifies the notion of proposition with that of type and the notion of proof of a proposition with that of element of a type. It is then possible to transform the type system into a proof system for intuitionistic logic. A proposition $A$ is intuitionistically true only when we have a proof for it, that is, if there is an element of type $A$. The proposition $A \& B$ is true only when we have both a proof (element) of proposition (type) $A$ and a proof (element) of proposition (type) $B$. Thus, the type $A \times B$ corresponds to the intuitionistic proposition $A \& B$. Interpreting the judgement $A \in T$ as $A$ proposition and the judgement $a \in A$ as $a$ true, the type rules of $\&$ become proof rules for intuitionistic $\&$:

$$A \& B =_{def} A \times B,$$

$\&$-formation

$$\frac{A, B \in \text{proposition}}{A \& B \in \text{proposition}}$$

$\&$-introduction

$$\frac{A \text{ true} \quad B \text{ true}}{A \& B \text{ true}}$$

$\&$-elimination

$$\frac{A \& B \text{ true}}{A \text{ true}} \quad \frac{A \& B \text{ true}}{B \text{ true}}$$

See [7] for the complete proof rules of intuitionistic logic.

A similar identification can be made in the context of programming languages [8,3]. The notion of program specification can be identified with that of type. That is, a program specification is the type consisting of all the programs that satisfy such a specification. Thus, programs are elements of types. The type system, in this case, becomes a program synthesis and verification system. Constable [3] uses the term programming logic for such a system.

We follow a similar approach to obtain a formalization of simulation. We adopt the following world-view for simulation modeling (see [4]). A new entity results from the synthesis of existing entities. That is, the creation process of a new entity uses as data the entities currently available in the system. We term concrescence this creation process and prehension the use of an existing entity as data in the creation of another entity\(^1\). In order to make precise the nature of this synthesis, we make the following identification: an entity is an element of a type. That is, entities correspond to well-typed $\lambda$-expressions. Activities can, in turn, be identified with subtypes. More precisely, an activity is the subtype consisting of all the entities of a type created (or engaging) in it.

As in the case of intuitionistic logic and programming languages, we are now in a position to use the tools of type theory for model description. The bridge from type theory into logic and programming languages is now available as well. That is, we can use the tools of type theory, logic, and programming languages for model construction, verification, and analysis. Entities and activities can be described by type introduction rules. Operations may be defined on entities of a given type using type elimination and equality rules.

To complete the picture for simulation languages, we define an event to be the creation of an entity. We assume that a certain amount of time passes between the moment when all the required entities become available for synthesis into a new entity and the moment the new entity is finally created in an activity. This time quantity may be stochastic. The specification of the time required to produce a new entity may be attached to the activity description or, in the case in which the entities of a given type are all created in the same activity, it may be attached to the type introduction rule.

For example, the rule

$$\frac{a \in A \quad b \in b}{c \in C \quad (\text{normal}(20,3))}$$

is interpreted to mean that a new entity $c$ of type $C$ is created $t$ time units after entities $a$ and $b$, of types $A$ and $B$ respectively, have become available, and that $t$ is distributed normally with a mean of 20 and a standard deviation of 3.

An activity system is defined by the collection of the relevant types of entities, the set of activities, and a world graph defined on the set of activities. To create a new entity, an activity may use as data the entities from any neighboring activity in the world graph. That is, an entity is created when all the entities required by its

---

\(^1\) Concrescence and prehension are terms used by A.N. Whitehead in his theory of actual entities. "Concrescence" is the "growth together" of existing entities to form a new "actual" entity; the new entity is said to "prehend" the existing entities used for its creation [9,4].
type introduction rule become present in neighboring activities.

In a simulation, samples of the entity creation times are used to trace the behavior of an activity system. Information functions may be defined to collect and report statistics.

\( \lambda \) is a simulation language based on this world-view [4].

4 Type inference

The type of an expression is determined from the types of its subexpressions. This recursive construction requires a type assumption for atomic expressions (identifiers). Let \( A \) denote the set of all assumption sets. The type of an expression is then a function of the assumptions made about its atomic subexpressions. We use

\[
A \vdash e \in \tau
\]
to denote the fact that expression \( e \) has type \( \tau \) when the assumptions in \( A \) are made. The following rules assign a type to every \( \lambda \sigma \) expression (entity):

(TAUT) \[
A \cup \{ e \in \tau \} \vdash e \in \tau
\]

(APP) \[
A \vdash e \in \tau, A \vdash e' \in \tau \rightarrow \tau' \\
A \vdash e' \in \tau'
\]

(\( \lambda \)-ABS) \[
A \cup \{ z \in \tau \} \vdash e \in \tau' \\
A \vdash \lambda z. e \in \tau \rightarrow \tau'
\]

(TUPLE) \[
A \vdash e_i \in \tau_i \ldots A \vdash e_n \in \tau_n \\
\{e_1, \ldots, e_n\} \in \tau_1 \times \cdots \times \tau_n
\]

The remaining expressions can be typed by adding the following to the assumption set:

\[
+ : \text{num} \times \text{num} \rightarrow \text{num} \\
- : \text{num} \times \text{num} \rightarrow \text{num} \\
* : \text{num} \times \text{num} \rightarrow \text{num} \\
/ : \text{num} \times \text{num} \rightarrow \text{num} \\
\text{\texttt{\!: bool \rightarrow bool}} \\
\text{\texttt{\&: bool \times bool \rightarrow bool}} \\
\text{\texttt{\texttt{[]: bool \times bool \rightarrow bool}}} \\
\text{\texttt{\texttt{[]: \tau_1 \times \cdots \times \tau_n \rightarrow \tau_1}}} \\
\text{\texttt{\texttt{[]: \tau_1 \times \cdots \times \tau_n \times \tau_1 \rightarrow \tau_1 \times \cdots \times \tau_n}}} \\
\ldots
\]

5 Type inference algorithm

The type of a \( \lambda \)-expression can be obtained using a type inference algorithm [8,2]. Based on the inference rules of a type system, it may be possible to construct an algorithm to determine the type of an expression by reconstructing its deduction.

The development is as follows. Consider the type system used in \( \lambda \sigma \). The type of an expression can be determined in a bottom-up fashion from the types of the subexpressions by repeatedly solving a set of simultaneous type equations. Type equations are solved by means of unification, that is, by finding a substitution of type expressions for type variables which will make the two sides of the equation identical [8]. Such a substitution is called a unifier. Let \( u(\tau_1, \tau_2) \) be the unifier of type expressions \( \tau_1 \) and \( \tau_2 \). Let \( S \) denote the set of all substitutions.

A recursive algorithm can be derived from the type inference rules as follows. Proceeding on the structure of \( \lambda \sigma \) expressions, we can construct the recursive type function

\[
\text{type} : \lambda \text{exp} \times A \rightarrow T \times S,
\]

which assigns a type \( \tau \) and a substitution \( S \) to an expression \( e \) in such a way that

\[
\text{type}(e, A) = (\tau, S)
\]

implies \( SA \vdash e \in \tau \).

In fact, it can be shown that if

\[
A \vdash e \in \tau',
\]

for some type \( \tau' \), then

\[
\text{type}(e, A) = (\tau, S),
\]

and there is a substitution \( S' \) such that

\[
S' \tau = \tau',
\]

that is, that the type \( \tau \) obtained by the algorithm is a most general type (see [8]).

The type function is defined as follows

\[
\text{type}(e, A \cup \{ e \in \tau \}) = (\tau, \emptyset) \\
\text{type}(e', A) = (US_2 \tau', US_2 S_1)
\]

where

\[
\text{type}(e', A) = (S \tau \rightarrow \tau', S_1), \text{type}(e, S A) = (\tau'', S_2),\]

and \( u(S_2 \tau, \tau'') = U \)

\[
\text{type}(\lambda e. A) = (\tau \rightarrow \tau', S)
\]

where \( \text{type}(e, A \cup \{ z \in \tau \}) = (\tau', S) \)

\[
\text{type}([e_1, e_2, \ldots, e_n], A) = (S_1 \ldots S_2 \tau_1 \times S_n \ldots S_3 \tau_2 \times \cdots \times \tau_n, S_n \cdots S_1)
\]

where

\[
\text{type}(e_1, A) = (\tau_1, S_1), \text{type}(e_2, S_1 A) = (\tau_2, S_2), \\
\ldots, \text{type}(e_n, S_{n-1} \cdots S_1 A) = (\tau_n, S_n)
\]
Example 1: Type inference.
To determine the type of
\[ e = [\lambda x, 1], \]
assuming that the type of 1 is \( \text{int} \), we let the type of \( e \) be \( \alpha_0 \). Now, letting the types of \( \lambda x, x \) and 1 be \( \alpha_1 \) and \( \alpha_2 \) respectively, and by noting that in this type system, pairs are of type \( A \times B \), we determine \( \alpha_0 \) to be \( \alpha_1 \times \alpha_2 \). Proceeding recursively, we eventually find that the types of \( \lambda x, x \) and 1 are actually \( \alpha_3 \rightarrow \alpha_3 \) and \( \text{int} \) respectively. Thus, we determine that \( \alpha_1 = \alpha_3 \rightarrow \alpha_3 \) and \( \alpha_2 = \text{int} \) and, finally, that
\[ \alpha_0 = (\alpha_3 \rightarrow \alpha_3) \times \text{int}. \]

6 Verification operations
In this section we describe several methods of model verification and analysis based on type inference. These methods are extensions of the unification-based type inference algorithm given in the previous section.

We present methods for the following model analysis and verification operations.

1. Type check all expressions.
2. Verify that all premises of rules in an activity can potentially be satisfied by its neighboring activities.
3. Verify that the entities created in an activity may be prehended by at least one neighbor.
4. Given an entity together with a set of assumptions for its free variables, determine the activities in which the entity could be created.

6.1 Expression type checking
This operation ensures that all expressions used to describe a \( \lambda \) model have, in fact, valid types. A type inferred for an expression may be kept for future reference in case the expression is used elsewhere. This is particularly important for expressions which have been abbreviated by an identifier, in order to determine the types of the expressions in which the identifier occurs.

Algorithm 1: Verify all the expressions in a model.
Input: A \( \lambda \sigma \) model \( M \)
Output: Successful termination if all expressions are type-error free

Method:
for each expression \( e \) in any component of \( M \) do
    Determine a set \( A \) of type assumptions for \( \text{freevars}(e) \);
    case 1: A type \( \tau \) for \( e \) is given.
        Let \( \tau' = \text{type}(e,A) \);
        if \( -(u(\tau, \tau') = \_2) \) then fail;
    case 2: The type for \( e \) is not specified.
        Let \( \tau = \text{type}(e,A) \);
        if \( \tau \) exists then
            declare it as the type of \( e \) assuming \( A \) for future reference
        else fail;
    end-case;
end-for.

Example 2: Verifying operation application.
Suppose that entities are created in activity \( A_1 \) by means of the rule
\[ f(g(a, b)) \in \tau_3, \]
where \( f \) and \( g \) have been previously defined.
By means of type inference we can determine that
\[ \{a \in \tau_1, b \in \tau_2 \} \vdash (a, b) \in \tau_3, \]
and that, therefore, the types of \( g \) and \( f \) must be \( \tau_1 \times \tau_2 \rightarrow \alpha \) and \( \alpha \rightarrow \tau_3 \), respectively for some type \( \alpha \).
It must now be verified that the types given in the definitions of \( f \) and \( g \) are unifiable to the types just inferred for them. This would not be the case if, for example, \( f \) were an operation that expected an integer as input, while \( g \) produced a pair of integers.

6.2 Premise satisfaction
An entity will not be created unless all the premises in the introduction rules that create it have been satisfied. This will not happen if, for each premise of the form \( e \in \tau \), there is not at least one neighboring activity that can potentially create an entity of type \( \tau' \), unifiable with \( \tau \), to satisfy it.

Algorithm 2: Verify that all premises can be satisfied by a neighbor.
Input: A $\lambda\sigma$ model $M$

Output: Successful termination if all activities can potentiallyprehend all required entities

Method:
for each activity $Act \in M$ do
  for each rule $RI \in Act$ do
    for each premise $e \in \tau$ in $RI$ do
      if there is a neighbor of $Act$ which can create an entity of type $\tau'$ such that $u(\tau, \tau') = -$ then continue
      else fail
  endfor.
endfor.

6.3 Potential entity prehension

This operation is a natural complement of the previous one. If an activity is not terminal, that is, if it is related as a neighbor in the direction of other activities, it is useful to verify that an entity created in the activity has a possible use in at least one neighbor. In this way, it can be verified that the entities created in an activity have the potential to flow out of it and into others.

Algorithm 3: Verify that all entities in activities with neighbors can be prehended by at least one neighbor.

Input: A $\lambda\sigma$ model $M$

Output: Successful termination if every entity created in an activity can potentially be prehended by at least one neighboring activity

Method:
for each activity $Act \in M$ do
  Let $\tau$ be the type of entities in $Act$
  for each rule $RI \in Act$ do
    if there is a neighbor of $Act$ with a rule requiring an entity of type $\tau'$ such that $u(\tau, \tau') = -$ then continue
    else fail
  endfor.
endfor.

6.4 Activities creating an entity

The following verification operation determines the set of activities in a $\lambda\sigma$ model which can possibly create a given entity.

The algorithm proceeds by examining the rules defined for a $\lambda\sigma$ model. The type of the given entity (which can be inferred as explained previously) is compared to the type of the entity created in the conclusion of the rule. If the two types unify, then all the activities using the rule to create new entities are added to the answer.

Algorithm 4: Determine the set of activities that can create a given entity.

Input: A $\lambda\sigma$ model $M$, a $\lambda$-expression $e$, and a set $A$ of assumptions for freevars ($e$)

Output: A set $S$ of activities which have the capability to create $e$

Method:
Let $\tau = \text{type}(e, A)$;
Let $S = \emptyset$;
for each rule $RI \in M$ do
  Let $e'$ be the expression and type specified in the conclusion of $RI$;
  if $u(\tau, \tau') = -$ then
    for each activity $Act \in M$ such that $RI \in Act$ do $S = S \cup \{Act\}$
  endfor.
endfor.

Example 3: Verifying entity creation and premise satisfaction.

Consider a machine-shop model. Suppose that we need to determine which activities create entities of the form

$$(m, (o_1, o_2), i),$$

where $m$ is a machine, $o_1$ and $o_2$ are operators, and $i$ is an item. Suppose that entities are created in activity $\text{order-accepted}$ according to the rule

$$(c, i) \in \text{order} \quad c \in \text{customer} \quad i \in \text{item} \quad \frac{(c, i) \in \text{order}}{(c, i) \in \text{order}},$$

that entities are created in activity $\text{team-ready}$ according to the rule

$$o_1, o_2 \in \text{operator} \quad \frac{(o_1, o_2) \in \text{team}}{(o_1, o_2) \in \text{team}},$$

that entities are created in activity $\text{job-assigned}$ according to the rule

$$m \in \text{machine} \quad t \in \text{team} \quad \frac{(c, i) \in \text{order}}{(m, t, i) \in \text{job}}.$$
and that entities are created in activity machine-free according to the rules
\[(m, t, i) \in \text{job}\]
\[m \in \text{machine}\]
\[m_1, \ldots, m_{10} \in \text{machine}\]
(there are initially 10 free machines in the shop).

Assuming that these activities are related in the model as shown in Figure 1, it is possible to verify that all entities required to create a new entity in activity job-assigned can potentially be created in neighboring activities by unifying the types of the entities created in these with the types of the entities required in the rule used in job-assigned.

Now, since
\[
\{o_1, o_2 \in \text{operator}\} \vdash (o_1, o_2) \in \text{operator} \times \text{operator},
\]
it is also possible to determine that
\[
\text{team} = \text{operator} \times \text{operator},
\]
and that therefore
\[
\text{job} = \text{machine} \times (\text{operator} \times \text{operator}) \times \text{item},
\]
which is the type of
\[(m_1(o_1, o_2), t).
\]
Job-assigned is then one of the activities which produce entities of the desired form.

7 Conclusion

In this paper we have described model verification in the \(\lambda \sigma\) simulation language. We have shown that, as a result of the foundation of \(\lambda \sigma\) on the typed \(\lambda\)-calculus, it is possible to use type inference algorithms for model verification and analysis.

Future work will involve extending the type system used for \(\lambda \sigma\) with constructs that increase the expressiveness of the language [4,7,2] and for which a type inference algorithm exists.

References