BabyJ: From Object Based to Class Based Programming via Types *

Christopher Anderson, Sophia Drossopoulou

Imperial College of Science, Technology and Medicine

Abstract. The popularity of object based languages has risen dramatically in recent years, owing to their use as scripts in HTML pages, or for interfacing with databases. A characteristic of these languages is interpretation and weak typing. Weak typing allows rapid prototyping but also runtime type errors. Therefore, it is desirable at a later development stage to have some assurances that your program will not go wrong.
Class based programming languages usually have static typing and thus give more robust programs. However, owing to the rigidity of the type system prototyping can be difficult.
We propose a language, BabyJ, that combines the benefits of object based programming and types. BabyJ allows rapid "typeless" development and also gives the programmer the opportunity to incrementally annotate the program with type information. Once fully typed, a BabyJ program can be converted to an equivalent Java program, that can be developed further.
This work is a continuation of previous work on BabyJ, in particular we have substantially improved the translation from BabyJ to Java, so that the code produced is in a class based style. Therefore, further development of the translated program is easier.

1 Introduction

Object based scripting languages have gained popularity over the past years. This is partly due to the World Wide Web, with scripting languages such as JavaScript[12] being used on the client side within web pages and on the server side with Python[17]. Their flexibility allows for rapid prototyping and quick software development. However, this flexibility comes at a price: due to their weak types, runtime type errors can occur.

Class based programming offers the possibility of creating more robust programs, owed in part to the static type systems of these languages. A notable exception is Smalltalk[10] which is class based but weakly typed. Classes and types in object oriented programming languages have overlapping, but not identical roles: Classes organize the code, and describe the behaviour of objects, whereas types describe the *interface* of objects. Although it has been successfully argued that classes and types should be distinguished [7], most commercial

* Work partly supported by DART, European Commission Research Directorates, IST-01-6-1A

programming languages (Java[11], C++[16], C-sharp[14] etc.) amalgamate these into one, and classes play the roles of types as well. In this work we follow that approach too, and attach types to functions that create objects.

We want to harness the positive aspects of both paradigms, that is, to combine rapid prototypical development with the robustness of a static type system.

In order to achieve this, we suggest that object based programs should be incrementally annotated with type information, type checked, and then translated to class based programs. In more detail, we suggest that one starts with an object based language to initially write programs without types, thus prototyping ideas. Type information can then be added in stages until the program is fully typed. Naturally, the program may need to be modified in places in order to be type correct. When fully typed, it can be *automatically* converted to an equivalent class based program. Therefore, the programmer benefits from rapid prototyping and ends up with a statically typed, class based program. This program then can be developed further or interfaced with other class based programs. Furthermore, compilation will bring some performance benefits.

We describe our approach through the languages BabyJ and BabyJ^T and a translation to a class based language, Java$_0$. BabyJ is inspired by JavaScript[1] and forms a small subset of it. As with JavaScript, BabyJ uses functions to create objects and allows dynamic addition of members to objects. BabyJ^T is a typed version of BabyJ. Types are introduced through constructors. There is also a permissive type, $*$, that allows any use. The programmer can use $*$ to initially annotate the program, and then incrementally replace occurrences of $*$ with types. Once fully typed, the program may be translated to an equivalent Java$_0$ class based program.

This paper is organized as follows: in section 2 we describe BabyJ. In section 3 we show how a BabyJ program can be incrementally typed and informally introduce BabyJ^T. In section 4 we give the typing of BabyJ^T and some properties of the type system. In section 5 we show how BabyJ^T programs can be translated into Java$_0$. In section 6 we compare our proposal to other approaches. In section 7 we discuss the limits of our approach and future work.

2 BabyJ

BabyJ is an object based language inspired by JavaScript. It includes the following JavaScript features:

1. Functions are used to create objects
2. Functions can be aliased and used as members of objects
3. Members can be added to objects dynamically

These features were chosen because, 1 represents the way objects are created in JavaScript, 2 is the way objects acquire methods, and 3 gives flexibility to programs. BabyJ does not include the following JavaScript features:

[1] Note that there is an international standard for JavaScript: ECMAScript

1. Strings as expressions
2. Libraries of functions
3. Native calls
4. Global `this` (through a global object)
5. Dynamic variable creation
6. Functions as objects
7. Dynamic removal of members
8. Delegation and prototyping

These features were omitted because 1,2 and 3 are not central to the paradigm, while 5,6 and 7 are too difficult to support in a statically typed language, and 8 will be considered when we extend the system in the future, see Section 7. We can write the introductory examples from [9] in BabyJ assuming, (2), integers, floats, strings etc.

Note, we allowed assignment to members of objects, therefore, trivially also allowed assignment of function members. However, we do not allow assignment to function identifiers.

2.1 BabyJ Example

In Figure 1 we give an example that describes people, money and marriage. People have a name and money and may marry other people. We define functions `Person`, `setAmount` and `Marry`. The code preceeded by the comment `//Main` is the entry point to the program.

The function `Person` is used to create two objects `becks` and `posh` on lines (2) and (3). The body of `Person` contains a sequence of assignments to the receiver `this`. When a function is used in the context of object creation through `new`, the receiver `this` is bound to an empty object at a new address in memory. Note, that as in JavaScript, assignment to a member of an object will create that member if it does not already exist within the object. Objects acquire methods through assignment of function identifiers to their members, as on line (1). Therefore, execution of `new Person` returns an object with three members: `name, money`, and `setMoney`.

There are two ways of invoking functions in BabyJ: either through an instance of an object, or globally. Consider now the function `setAmount`, which updates the `money` attribute of the receiver by `amount`. This demonstrates calling a function through an instance of an object, the receiver, `this` is bound to that object. For example, `becks.setMoney(30)`, will bind `becks` to `this` when `setAmount` is executed. As the member `money` already exists within `becks`, it will be updated rather than created.

The function `Marry`, which is used in the main body to marry `becks` and `posh`, demonstrates calling a function globally. Note also, that within the body of `Marry` there is no reference to `this`. This is because there is no receiver and hence `this` is undefined. Therefore, any references to `this` within a function invoked globally will produce a runtime error. Again, `Marry` demonstrates automatic creation of attributes through assignment, as in this example, `becks` and `posh` had no `partner` member prior to execution of `Marry`.

```
function Person(name,money) {
  this.name = name;
  this.money = money;
  this.setMoney = setAmount;         // (1)
  this
}

function setAmount(amount) {
  this.money += amount
}

function Marry(p1,p2) {
  p1.partner = p2;
  p2.partner = p1
}
//Main
var posh = new Person("Posh",100);   // (2)
var becks = new Person("Becks",80);  // (3)
Marry(posh,becks);
becks.partner.setMoney(20);
posh.money
```

Fig. 1. People example program

2.2 BabyJ Syntax

We give the syntax of BabyJ in Figure 2. P stands for a sequence of function declarations. We implicitly require functions to be unique, $\vdash \mathrm{P} \diamond_u$, as defined in Section A.3, and then use P as a mapping $\mathrm{P} :: \mathit{FuncID} \rightarrow \mathit{FuncDecl}$.

Note that the example in Figure 1 uses a more liberal syntax than that in Figure 2. In particular, it allows functions with more than one parameter, integers and a main body of code.

2.3 BabyJ Operational Semantics

Figure 3 gives a structural operational semantics that rewrites tuples of expressions, heaps and stacks into tuples of values, heaps and stacks in the context of a program, P. The signature of the rewriting relation $\rightsquigarrow$ is:

$$\rightsquigarrow : \mathit{Program} \rightarrow \mathit{Exp} \times \mathit{Heap} \times \mathit{Stack} \rightarrow (\mathit{Val} \cup \mathit{Dev}) \times \mathit{Heap} \times \mathit{Stack}$$

The heap maps addresses to objects, where addresses, *Addr*, are $\iota_0, ..\iota_n ...$ Objects are finite mappings from member identifiers to values, indicated by $\xrightarrow[fin]{}$, we use $\mathcal{U}df$ for undefined. The stack maps `this` to an address and variables to values.

$$\begin{array}{lll} \mathsf{H} \in \mathit{Heap} & = \mathit{Addr} \xrightarrow[fin]{} \mathit{Obj} \\ \mathsf{S} \in \mathit{Stack} & = (\{\mathtt{this}\} \rightarrow \mathit{Addr}) \cup (\{\mathtt{x}, \mathtt{y}\} \rightarrow \mathit{Val}) \\ \mathsf{v} \in \mathit{Val} & = \{\mathtt{null}\} \cup \mathit{FuncID} \cup \mathit{Addr} \\ \mathsf{dv} \in \mathit{Dev} & = \{\mathtt{nullPntrExc}, \mathtt{stuckErr}\} \\ \mathit{Obj} & = \mathit{MemberID} \xrightarrow[fin]{} \mathit{Val} \end{array}$$

P ∈ *Program*	::=	D*	
B ∈ *Body*	::=	`var y; e`	
D ∈ *FuncDecl*	::=	`function f (x) {`B`}`	
e ∈ *Exp*	::=	`this`	receiver
		`f`	function identifier
		`x`	parameter
		`y`	local variable
		`new f(e)`	object creation
		`null`	null
		`e; e`	sequence
		`e.m(e)`	member call
		`e.m`	member select
		`f(e)`	global call
		`v = e`	assignment
v ∈ *Var*	::=	`x` | `y` | `e.m`	

f ∈ *FuncID*
m ∈ *MemberID*
FuncID ∩ *MemberID* = ∅

Fig. 2. Syntax of BabyJ

Unlike other object calculi such as [1, 6], method bodies are not stored inside objects. Instead, we store the function identifier corresponding to the method. This allows aliasing to the function members. We treat P as "global", therefore $\mathsf{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'$ is shorthand for $\mathsf{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow_P \mathsf{v}, \mathsf{H}', \mathsf{S}'$. We use heap update, $\mathsf{H}' = \mathsf{H}\{\iota.\mathsf{m} \lhd \mathsf{v}\}$, where H' is identical to H except that in ι member m is overridden by v. If ι does not already have a member called m, then such a member will be added. Heap update is defined in Section A.1

Figure 3 defines the operational semantics without generation and propagation of exceptions (these are given in Section E). We now discuss the most interesting rules: *(Var)*, *(Member-select)*, *(New)*,*(Param-ass)*, *(Local-ass)*, *(Member-call)* and *(Global-call)*.

In *(Var)* the receiver (`this`) or parameter (`x`) or local variable (`y`) are looked up in the stack, and heap and stack are unmodified. A function name, (`f`) is not looked up, as it is a value.

In *(Member-select)* member m is looked up in the receiver ι (obtained by evaluation of e) in the heap. If m is not found in ι, then execution is stuck.

In *(New)* we execute the body of function f (looked up in P) with a stack that maps `this` to a fresh address that points to an empty object, formal parameter (`x`) to the value obtained by execution of the actual parameter and local variable (`y`) to $\mathcal{U}df$.

In *(Param-ass)* and *(Local-ass)* we replace the value of `x` and `y` respectively, in the stack with the value obtained by execution of `e`. Unlike JavaScript, we only have one local variable declared at the beginning of a function and one function parameter.

In *(Member-call)* we obtain the function definition by looking up the value of member `m` in the receiver (obtained by evaluation of `e`) in P. We execute the body with a stack similar to that for *(New)* except that `this` points to the receiver.

Global calls *(Global-call)* have the same format as member calls, expect that there is no receiver, therefore `f` is looked up directly in P and the stack maps `this` to $\mathcal{U}df$.

3 Adding Type Information - An Example

In Figure 4 we return to the example of Section 2.1 and incrementally add type information. We make the following observations:

- The function's return type is given after the keyword `function`.
- The types of formal parameters and variables are given after their introduction.
- Function bodies start by declaring the type of the receiver, `this`.
- Types include the special type $*$, which allows any operation.

All types in Figure 4 are $*$'s, therefore although the program type checks, this does not give any guarantees of run-time safety. We want to replace all occurrences of $*$. We therefore need to introduce types, and use them to annotate the program. Types are introduced by constructors. The constructor name is the name of the type. The structure of that type is given in the *type descriptor* following `this` at the beginning of the body of the constructor. For example, `Person`, is declared to be a constructor. The first line in its body gives the structure of that type, and says that it has members `name, money` and `setMoney`, whose type is still unspecified:

```
constructor Person(name:*,money:*) {
  this:[name:*,money:*,setMoney:*];
  this.name = name;
  this.money = money;
  this.setMoney = setAmount;
  this
}
```

The type `Person` may be used from now on. Member functions have to specify the type of their receiver. For example, `setAmount` has receiver type `Person`:

```
function * setAmount(amount:*) {
  this:Person;
  this.money += amount
}
```

(*Var*)

$$\frac{}{\mathtt{this}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{S}(\mathtt{this}), \mathsf{H}, \mathsf{S}}$$
$$\mathtt{x}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{S}(\mathtt{x}), \mathsf{H}, \mathsf{S}$$
$$\mathtt{y}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{S}(\mathtt{y}), \mathsf{H}, \mathsf{S}$$
$$\mathsf{f}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{f}, \mathsf{H}, \mathsf{S}$$

(*Member-select*)

$$\frac{\mathtt{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow \iota, \mathsf{H}', \mathsf{S}'}{\mathtt{e.m}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{H}'(\iota)(\mathtt{m}), \mathsf{H}', \mathsf{S}'}$$

(*Seq*)

$$\frac{\mathtt{e}_1, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}', \mathsf{H}_1, \mathsf{S}_1 \quad \mathtt{e}_2, \mathsf{H}_1, \mathsf{S}_1 \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'}{\mathtt{e}_1; \mathtt{e}_2, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'}$$

(*New*)

$$\frac{\begin{array}{l}\mathtt{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}', \mathsf{H}_1, \mathsf{S}' \\ \mathrm{P}(\mathsf{f}) = \mathtt{function\ f(x)\{var\ y;\ e'\}} \\ \iota \text{ is new in } \mathsf{H}_1 \text{ and } \mathsf{H}_2 = \mathsf{H}_1\{\iota \mapsto [\![\ ]\!]\} \\ \mathtt{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}' \mathtt{y} \mapsto \mathcal{U}df, \} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}''\end{array}}{\mathtt{new\ f(e)}, \mathsf{H}, \mathsf{S} \rightsquigarrow \iota, \mathsf{H}', \mathsf{S}'}$$

(*Member-ass*)

$$\frac{\begin{array}{l}\mathtt{e}_1, \mathsf{H}, \mathsf{S} \rightsquigarrow \iota, \mathsf{H}_1, \mathsf{S}_1 \\ \mathtt{e}_2, \mathsf{H}_1, \mathsf{S}_1 \rightsquigarrow \mathsf{v}, \mathsf{H}_2, \mathsf{S}' \\ \mathsf{H}' = \mathsf{H}_2\{\iota.\mathtt{m} \lhd \mathsf{v}\}\end{array}}{\mathtt{e}_1.\mathtt{m} = \mathtt{e}_2, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'}$$

(*Param-ass*)

$$\frac{\begin{array}{l}\mathtt{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'' \\ \mathsf{S}' = \{\mathtt{this} \mapsto \mathsf{S}''(\mathtt{this}), \mathtt{x} \mapsto \mathsf{v}, \mathtt{y} \mapsto \mathsf{S}''(\mathtt{y})\}\end{array}}{\mathtt{x} = \mathtt{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'}$$

(*Local-ass*)

$$\frac{\begin{array}{l}\mathtt{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'' \\ \mathsf{S}' = \{\mathtt{this} \mapsto \mathsf{S}''(\mathtt{this}), \mathtt{x} \mapsto \mathsf{S}''(\mathtt{x}), \mathtt{y} \mapsto \mathsf{v}\}\end{array}}{\mathtt{y} = \mathtt{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'}$$

(*Member-call*)

$$\frac{\begin{array}{l}\mathtt{e}_1, \mathsf{H}, \mathsf{S} \rightsquigarrow \iota, \mathsf{H}_1, \mathsf{S}_1 \\ \mathtt{e}_2, \mathsf{H}_1, \mathsf{S}_1 \rightsquigarrow \mathsf{v}', \mathsf{H}_2, \mathsf{S}' \\ \mathsf{H}_2(\iota)(\mathtt{m}) = \mathsf{f} \\ \mathrm{P}(\mathsf{f}) = \mathtt{function\ f(x)\{var\ y;\ e'\}} \\ \mathtt{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}' \mathtt{y} \mapsto \mathcal{U}df, \} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}''\end{array}}{\mathtt{e}_1.\mathtt{m}(\mathtt{e}_2), \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'}$$

(*Global-call*)

$$\frac{\begin{array}{l}\mathtt{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}', \mathsf{H}_1, \mathsf{S}' \\ \mathrm{P}(\mathsf{f}) = \mathtt{function\ f(x)\{var\ y;\ e'\}} \\ \mathtt{e}', \mathsf{H}_1, \{\mathtt{this} \mapsto \mathcal{U}df, \mathtt{x} \mapsto \mathsf{v} \mathtt{y} \mapsto \mathcal{U}df\} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}''\end{array}}{\mathsf{f}(\mathtt{e}), \mathsf{H}, \mathsf{S} \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'}$$

Fig. 3. Operational Semantics of BabyJ

```
function * Person(name:*,money:*) {
  this:*
  this.name = name;
  this.money = money;
  this.setMoney = setAmount;
  this
}

function * setAmount(amount:*) {
  this:*
  this.money += amount
}

function * Marry(p1:*,p2:*) {
  this:*
  p1.partner = p2;
  p2.partner = p1
}
//Main
var becks:* = new Person("Becks",100);
var posh:* = new Person("Posh",80);
Marry(becks,posh);
becks.partner.setMoney(20);
posh.money
```

Fig. 4. Annotated people example

```
constructor Person(name:string,money:int) {
  this:[name:string,money:int,setMoney:(Person,int,int),partner:Person]
  this.name = name;
  this.money = money;
  this.setMoney = setAmount;
  this
}

function int setAmount(amount:int) {
  this:Person;
  this.money += amount
}

global Person Marry(p1:Person,p2:Person) {
  p1.partner = p2;
  p2.partner = p1
}
//Main
var becks:Person = new Person("Becks",100);
var posh:Person = new Person("Posh",80);
Marry(becks,posh);
becks.partner.setMoney(20);
posh.money
```

Fig. 5. Fully type people example

Global functions are those which are not applied to objects, and are specified as `global`. Eg, `Money` is a global function with parameters and return type `Person`:

```
global Person Marry(p1:Person,p2:Person) {
  p1.partner = p2;
  p2.partner = p1
}
```

At this point we will obtain a type error as `Person` has no member `partner`. We therefore add to the type descriptor of `Person` the member `partner` of type `Person`, *and* modify the body of the constructor to create a `partner` member, `this.partner = null`. We now replace the occurrences of $*$ in the type descriptor for `Person`, assuming primitive types `int` for integers and `string` for strings: `this:[name:string, money:int,setMoney:(Person,int,int), partner:Person]`. Note, that the type `(Person,int,int)` is a member function type whose receiver is of type `Person` and input and output are of type `int`. Now all the assignments to members of `Person` can be checked in functions `setAmount` and `Marry`. We now replace the rest of the $*$'s in the program. The fully annotated program without any $*$ is shown in Figure 5. For the rest of this paper we shall refer to the program in Figure 5, excluding the main body, as $\mathbb{P}_1$.

4 BabyJ^T - Incrementally typed BabyJ

BabyJ^T is a typed version of BabyJ allowing the programmer to incrementally add type information to a program.

4.1 Syntax of BabyJ^T

The syntax of BabyJ is given in Figure 6. There are three kinds of function in BabyJ^T: constructor, member function and global function. Constructors are used to create objects as expected in a class based object oriented language. Member functions are used as methods of objects. Global functions are present in JavaScript and are useful when a body of code does not belong to a particular object but is required by the programmer.

4.2 Types in BabyJ^T

The design of the types in BabyJ^{T}was driven by the design of BabyJ. The syntax of types is shown in Figure 6. Constructors describe the behaviour of objects and also give the interface to the object through a type descriptor, $TypeDesc$. Hence, constructor names, as well as the permissive type, $*$, make up the set of types *ObjectType*. Therefore, constructors have the same role as a Java class and constructor. Because members of objects can be functions and they can be reassigned, we require an explicit function type, *FunctionType*, for the identifiers of member functions. *FunctionType* has the form $(\mathsf{ts}_1, \mathsf{t}_2, \mathsf{t}_3)$ where ts_1 is the

receiver and input $\mathtt{t}_2$ and output $\mathtt{t}_3$. The syntax of *FunctionType* allows higher order types in that member functions can be passed as arguments, or returned from functions. However, the receiver must be an *ObjectType* as the function will be a member of some constructor used to create objects.

$\mathbb{P} \in$ *TypedProgram* ::= $\mathbb{D}^*$
$\mathbb{B} \in$ *TypedBody* ::= `var y : t ; e`
$\mathbb{D} \in$ *TypedFuncDecl* ::= *MemberFunction* | *GlobalFunction* | *CtrFunction*
MemberFunction ::= `function t f (x: t) { this : ts; B }`
GlobalFunction ::= `global t f (x: t) { B }`
CtrFunction ::= `constructor f (x: t) { this : td; B }`

Where type are:

$\mathsf{t} \in$ *ObjectType* $\cup$ *FunctionType*
$\mathsf{ts} \in$ *ObjectType* ::= $\mathsf{f} \mid *$
$\mathsf{tf} \in$ *FunctionType* ::= $(\mathsf{ts}, \mathsf{t}, \mathsf{t})$
$\mathsf{td} \in$ *TypeDesc* ::= $[\ (\mathsf{m} : \mathsf{t})^*\ ]$

Fig. 6. Syntax of types in BabyJ^T

4.3 Typing of Expressions

Typing an expression e in the context of a program $\mathbb{P}$, and environment Γ has the form $\mathbb{P}, \Gamma \vdash \mathsf{e} : \mathsf{t}$. Environments, ranged over by meta variable Γ, are mappings from variables to types. They have the form $\{\mathtt{this} : \mathsf{t}, \mathtt{x} : \mathsf{t}', \mathtt{y} : \mathsf{t}''\}$.

For example, with $\mathbb{P}_1$ from section 3, environment such that $\Gamma(\mathtt{becks}) = \mathtt{Person}$, we have:

$$\mathbb{P}_1, \Gamma \vdash \mathtt{becks} = \mathtt{new\ Person}(...) : \mathtt{Person}$$

The type rules are given in Figure 7. We use the relation $\mathsf{t} \approx \mathsf{t}'$ which holds if t and t' are the same identifier or one of them is $*$, and the function $\mathcal{D}(\mathbb{P},\mathsf{ts},\mathsf{m})$ which extracts the type of member m from the type descriptor for t in P.

The first rules, i.e. *(Var)* to *(Global Call)*, describe typing as would be expected in a mini object based language, i.e. the creation of new objects, selection of members, call of member function etc. The use of the relation $\mathsf{t} \approx \mathsf{t}'$ allows expressions of type $*$ to be substituted in any context. For example, if $\Gamma(\mathtt{x1}) = \mathtt{Person}$; $\Gamma(\mathtt{x2}) = *$, then

$$\mathbb{P}_1, \Gamma \vdash \mathtt{Marry}(\mathtt{x1}, \mathtt{x2}) : \mathtt{Person}$$

The next rule *(Member-func)* gives a type to member functions. We do not give types to global functions and constructors because the former can only be

Object Based

(*Var*)

$$\frac{}{\mathbb{P}, \Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}$$
$$\mathbb{P}, \Gamma \vdash \mathtt{y} : \Gamma(\mathtt{y})$$
$$\mathbb{P}, \Gamma \vdash \mathtt{this} : \Gamma(\mathtt{this})$$

(*Null*)

$$\frac{}{\mathbb{P}, \Gamma \vdash \mathtt{null} : \mathtt{ts}}$$

(*Seq*)

$$\frac{\mathbb{P}, \Gamma \vdash \mathtt{e}_1 : \mathtt{t}' \quad \mathbb{P}, \Gamma \vdash \mathtt{e}_2 : \mathtt{t}}{\mathbb{P}, \Gamma \vdash \mathtt{e}_1; \mathtt{e}_2 : \mathtt{t}}$$

(*New-cons*)

$$\frac{\mathbb{P}(\mathtt{f}) = \mathtt{constructor}\ \mathtt{f}(\mathtt{x} : \mathtt{t})\{...\} \quad \mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{t}' \quad \mathtt{t} \approx \mathtt{t}'}{\mathbb{P}, \Gamma \vdash \mathtt{new}\ \mathtt{f}(\mathtt{e}) : \mathtt{f}}$$

(*Member-sel*)

$$\frac{\mathbb{P}, \Gamma \vdash e : \mathtt{t}' \quad \mathcal{D}(\mathbb{P}, \mathtt{t}', \mathtt{m}) = \mathtt{t}}{\mathbb{P}, \Gamma \vdash \mathtt{e.m} : \mathtt{t}}$$

(*Var-ass*)

$$\frac{\mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{t} \quad \Gamma(\mathtt{x}) \approx \mathtt{t}}{\mathbb{P}, \Gamma \vdash \mathtt{x} = \mathtt{e} : \mathtt{t}}$$

(*Member-ass*)

$$\frac{\mathbb{P}, \Gamma \vdash \mathtt{e}_1 : \mathtt{t}' \quad \mathbb{P}, \Gamma \vdash \mathtt{e}_2 : \mathtt{t}'' \quad \mathcal{D}(\mathbb{P}, \mathtt{t}', \mathtt{m}) = \mathtt{t} \quad \mathtt{t} \approx \mathtt{t}''}{\mathbb{P}, \Gamma \vdash \mathtt{e}_1.\mathtt{m} = \mathtt{e}_2 : \mathtt{t}''}$$

(*Member-call*)

$$\frac{\mathbb{P}, \Gamma \vdash \mathtt{e}_1 : \mathtt{t}' \quad \mathbb{P}, \Gamma \vdash \mathtt{e}_2 : \mathtt{t}'_1 \quad \mathcal{D}(\mathbb{P}, \mathtt{t}', \mathtt{m}) = (\mathtt{t}', \mathtt{t}_1, \mathtt{t}'') \quad \mathtt{t}_1 \approx \mathtt{t}'_1}{\mathbb{P}, \Gamma \vdash \mathtt{e}_1.\mathtt{m}(\mathtt{e}_2) : \mathtt{t}''}$$

(*Global-call*)

$$\frac{\mathbb{P}(\mathtt{f}) = \mathtt{global}\ \mathtt{t}\ \mathtt{f}(\mathtt{x} : \mathtt{t}''))\{\ ...\} \quad \mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{t}' \quad \mathtt{t}'' \approx \mathtt{t}'}{\mathbb{P}, \Gamma \vdash \mathtt{f}(\mathtt{e}) : \mathtt{t}}$$

Member Functions

(*Member-func*)

$$\frac{\mathbb{P}(\mathtt{f}) = \mathtt{function}\ \mathtt{t}\ \mathtt{f}(\mathtt{x} : \mathtt{t}')\{\mathtt{this} : \mathtt{ts}; ...\}}{\mathbb{P}, \Gamma \vdash \mathtt{f} : (\mathtt{ts}, \mathtt{t}', \mathtt{t})}$$

Early Stages

(*Any-call*)

$$\frac{\mathbb{P}(\mathtt{f}) = \mathtt{function}\ \mathtt{t}\ \mathtt{f}(\mathtt{x} : \mathtt{t}'')\{\ \mathtt{this} : *; ...\} \quad \mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{t}' \quad \mathtt{t}'' \approx \mathtt{t}'}{\mathbb{P}, \Gamma \vdash \mathtt{f}(\mathtt{e}) : \mathtt{t}}$$

(*New-empty*)

$$\frac{\mathbb{P}(\mathtt{f}) = \mathtt{function}\ \mathtt{t}\ \mathtt{f}(\mathtt{x} : \mathtt{t}'')\{\ \mathtt{this} : *; ...\} \quad \mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{t}' \quad \mathtt{t}'' \approx \mathtt{t}'}{\mathbb{P}, \Gamma \vdash \mathtt{new}\ \mathtt{f}(\mathtt{e}) : *}$$

Fig. 7. Typing rules for expressions

invoked through a function identifier and the later through `new`. For example, if $\Gamma(\mathtt{x}) = \mathtt{Person}$, then

$$\mathbb{P}_1, \Gamma \vdash \mathtt{x.setAmount} = \mathtt{setAmount} : (\mathtt{Person}, \mathtt{int}, \mathtt{int})$$

Finally, the last two rules, *(Any-call)* and *(New-empty)* will be used at the early stages of adding types to the program. The first allows the use of a member function as a global function, as long as its receiver is still typed with $*$. This is useful when a function is used as both a member and a global function. The other rule *(New-empty)*, allows the use of a function that has not yet been designated to be a constructor, member or global function, to be used as a constructor.

4.4 Well-formed Programs

A program $\mathbb{P}$ is well-formed ($\vdash \mathbb{P}\diamond$), if all its functions are well formed ($\mathbb{P} \vdash \mathtt{f}\diamond$) and unique ($\vdash \mathbb{P}\diamond_u$). The definitions are given in Figure 8. Well formedness of member functions, global functions, and constructors is checked in an environment which maps the parameter $\mathtt{x}$ to $\mathtt{t}_1$ according to its declarations.

Furthermore, for constructors and member functions the environment maps `this` to the name of the constructor or the type given in the function body respectively. Functions have to have a body whose type is compatible with the declared return type, or for constructors the type of object they create.

Finally, constructor bodies have to start by giving values to all the members as checked by the function $\mathcal{F}ilter :: \mathtt{e} \rightarrow \mathit{MethID}$. We explain $\mathcal{F}ilter(\mathtt{e})$ in more detail in Section A.4. Any members of the type descriptor that are of function type must be of member function type where $(\mathtt{f}, \mathtt{t}', \mathtt{t}'')$. This ensures that methods belong to objects of the correct type.

4.5 Properties of the type system

We now give some properties of the type system that will be useful when showing soundness of the BabyJ^T type system. Strong types and functions are those that do not contain any occurrence of $*$. We say that a program, $\mathbb{P}$, is strongly typed, $\vdash \mathbb{P}\diamond_S$, if it is well-formed and each of its functions are strong. An environment, Γ, is strong, $\vdash \Gamma\diamond_S$, if it does not map any of its domain to $*$. Definitions are given in Section C.

4.6 Soundness

The judgement $\mathbb{P}, \mathsf{H} \vdash \mathtt{v} \lhd \mathtt{t}$ guarantees that value $\mathtt{v}$ conforms to type $\mathtt{t}$. In particular, it requires that if $\mathtt{v}$ is an address, ι, it points to an object that contains all the members defined in $\mathtt{t}$ and that members conform to their types. A typing function $\mathcal{T} :: \mathit{Addr} \rightarrow \mathit{FuncID}$, maps addresses in the heap to identifiers corresponding to constructors.

The judgement $\mathbb{P}, \Gamma, \mathcal{T} \vdash \mathsf{H}, \mathsf{S} \diamond$ guarantees that all objects conform to their types according to $\mathcal{T}$, and that the parameter, local variable and receiver are mapped to values that conform to their types in Γ.

$$\frac{\mathsf{f} = \texttt{function t f(x : t}_1\texttt{)}\ \{\texttt{this : ts; var y : t}'_1\texttt{; e;}\}\quad \mathbb{P}, \{\texttt{this} \mapsto \mathsf{ts}, \mathsf{x} \mapsto \mathsf{t}_1, \mathsf{y} \mapsto \mathsf{t}'_1\} \vdash \mathsf{e} : \mathsf{t}}{\mathbb{P} \vdash \mathsf{f}\diamond_s} \qquad \frac{\mathsf{f} = \texttt{global t f(x : t}_1\texttt{)}\ \{\texttt{var y : t}'_1\texttt{; e;}\}\quad \mathbb{P}, \{\mathsf{x} \mapsto \mathsf{t}_1, \mathsf{y} \mapsto \mathsf{t}'_1\} \vdash \mathsf{e} : \mathsf{t}}{\mathbb{P} \vdash \mathsf{f}\diamond_g}$$

$$\frac{\begin{array}{l}\mathsf{f} = \texttt{constructor f(x : t}_1\texttt{)}\ \{\texttt{this : [m}_1 \texttt{ : t}'_1 \ldots \texttt{m}_p \texttt{ : t}'_p\texttt{]; var y : t}''_1\texttt{; e;}\} \\ \forall\, \mathsf{t} \in \{\mathsf{t}'_1 \ldots \mathsf{t}'_p\} : \mathsf{t} = (\mathsf{t}', \_, \_) \implies \mathsf{t}' = \mathsf{f} \\ \mathcal{F}ilter(\mathsf{e}) = \{\mathsf{m}_1 \ldots \mathsf{m}_p\} \\ \mathbb{P}, \{\texttt{this} \mapsto \mathsf{f}, \mathsf{x} \mapsto \mathsf{t}_1, \mathsf{y} \mapsto \mathsf{t}''_1\} \vdash \mathsf{e} : \mathsf{f}\end{array}}{\mathbb{P} \vdash \mathsf{f}\diamond_c}$$

$$\frac{\mathbb{P} \vdash \mathsf{f}\diamond_s}{\mathbb{P} \vdash \mathsf{f}\diamond} \qquad \frac{\mathbb{P} \vdash \mathsf{f}\diamond_c}{\mathbb{P} \vdash \mathsf{f}\diamond} \qquad \frac{\mathbb{P} \vdash \mathsf{f}\diamond_g}{\mathbb{P} \vdash \mathsf{f}\diamond} \qquad \frac{\vdash \mathbb{P}\diamond_u \quad \forall \mathsf{f} \in \mathit{dom}(\mathbb{P}) : \mathbb{P} \vdash \mathsf{f}\diamond}{\vdash \mathbb{P}\diamond}$$

Fig. 8. Well-formed functions and programs

Types do not affect the execution of BabyJ^T programs. Therefore, we can "strip" the type information, using function $\mathcal{S}trip$, and use the operational semantics of BabyJ for BabyJ^T. Formal definitions can be found in Section C.

The type system is sound in the sense that a converging well-typed expression in the context of a strongly typed program and strong environment returns a value that agrees with the expression's type, or `nullPntrExc`, but is *never* stuck.

Theorem 1. *If* $\vdash \mathbb{P}\diamond_S$, *and* $\mathbb{P}, \Gamma, \mathcal{T} \vdash \mathsf{H}, \mathsf{S}\ \diamond$, *and* $\vdash \Gamma\diamond_S$, *and* $\mathbb{P}, \Gamma \vdash \mathsf{e} : \mathsf{t}$ *and* $\mathsf{e}, \mathsf{H}, \mathsf{S}$ *converges then*

- $\mathsf{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow_{\mathcal{S}trip(\mathbb{P})} \mathsf{v}, \mathsf{H}', \mathsf{S}'$, *and* $\mathbb{P}, \mathsf{H}' \vdash \mathsf{v} \lhd \mathsf{t}$, *and* $\mathbb{P}, \Gamma, \mathcal{T}' \vdash \mathsf{H}', \mathsf{S}'\ \diamond$, *and* $\mathcal{T}'$ *extends* $\mathcal{T}$,

or

- $\mathsf{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow_{\mathcal{S}trip(\mathbb{P})} \texttt{nullPntrExc}, \mathsf{H}', \mathsf{S}'$

As far as divergent expressions go, the theorem does not say anything. However, the operational semantics forces convergence for standard typing errors or access to members undefined in an object, see Section E.

5 From BabyJ to Java$_0$

We now present a translation from strongly typed BabyJ^T ($\vdash \mathbb{P}\diamond_S$) to Java$_0$[5], a small subset of Java. The translation function, $\llbracket\ \rrbracket$, is defined in Section D. Figure 9 gives the translation of Figure 5. The aim of the translation is to generate

programs in an object oriented style that are readable and maintainable. Each constructor is represented as a class that has a field for each member of the type descriptor.

One of the major challenges of the translation was representing BabyJ members. Namely, in BabyJ, as in Javascript, members may be functions and may be re-assigned. However, in Java_0 (and in Java) methods may not be re-assigned. In earlier versions of this work we mapped BabyJ^T function members onto functions objects, and fields of function type. Thus, assignment of function members in BabyJ^T was mapped in Java_0 as assignment of these fields to the function objects, and method call was mapped onto a method call of these objects where the receiver was passed as a parameter. This resulted in correct code which, however was not in the style expected in class based languages.

In the current paper we have a better solution: BabyJ^T member functions are mapped onto Java_0 methods. The method body contains a case statement which distinguishes between the various possible cases for the function - these are known because the candidates must have the same type as the function member. Furthermore, for each function member a field of type `int` keeps track of the latest assignment to function members (used in the case statement). The translation uses a one-to-one mapping, $\mathcal{M} :: FuncID \rightarrow \mathbb{N}$, from member function identifiers to integers. Therefore, BabyJ assignment to function members is represented in Java_0 through an `int` field, and function call is represented as method call. We thus obtain a Java_0 program written in an class based style.

5.1 Preservation of static and dynamic semantics

We now define some formal properties of the translation. We let meta variables R,T and γ range over Java_0 heaps, stacks, and environments respectively.

Theorem 2. *For any strongly typed BabyJ^T program $\vdash \mathbb{P} \diamond_S$, the translation $(\!|\mathbb{P}|\!)$ is well-formed in Java$_0$.*

We discuss the proof of this theorem in Section F. We now show that the semantics of expressions is preserved by the translation. We introduce a relation between values, $\mathcal{M} \vdash \mathsf{v} \approx_b \mathsf{v}'$ that expresses the fact that value v' is the translation of v, with respect to b, a bijection between addresses in H and R, and a mapping $\mathcal{M}$.

We now introduce a relation between pairs of heaps and stacks, $\mathbb{P}, \Gamma, \mathcal{M} \vdash \mathsf{H}, \mathsf{S} \approx_b \mathsf{R}, \mathsf{T}$ that expresses the fact that R contains the "translation" of the objects in store H and that T and S agree, w.r.t. typing, with their definitions. We define both relations in Section F. We now state the soundness theorem:

Theorem 3. *If $\vdash \mathbb{P} \diamond_S$, and $\vdash \Gamma \diamond_S$, $\mathbb{P}, \Gamma, \mathcal{T} \vdash \mathsf{H}, \mathsf{S} \diamond$, and $(\!|\mathbb{P}|\!)_{(\mathbb{P},\mathcal{M})}, (\!|\Gamma|\!) \vdash \mathsf{R}, \mathsf{T} \diamond$, and $\mathbb{P}, \Gamma, \mathcal{M} \vdash \mathsf{H}, \mathsf{S} \approx_b \mathsf{R}, \mathsf{T}$, and $\mathsf{e}, \mathsf{H}, \mathsf{S} \rightsquigarrow_{\mathcal{S}trip(\mathbb{P})} \mathsf{v}, \mathsf{H}', \mathsf{S}'$.*

Then there exists R',T' such that $\mathsf{e}, \mathsf{R}, \mathsf{T} \rightsquigarrow_{(\mathbb{P})} \mathsf{v}', \mathsf{R}', \mathsf{T}'$ and $\mathbb{P}, \Gamma, \mathcal{M} \vdash \mathsf{H}', \mathsf{S}' \approx_{b'} \mathsf{R}', \mathsf{T}'$, and $\mathcal{M} \vdash \mathsf{v} \approx_{b'} \mathsf{v}'$ and b' extends b.

Proof of the theorem by induction on the application of the rules defining the operational semantics.

```
class Person {
 String name;
 int Money;
 int setMoney;

 Person(String name,int money) {
   this.name = name;
   this.money = money;
   this.setMoney = 1;
 }
 int setMoney(int amount) {
   switch(setMoney) {
      case 1: {
       this.money += amount;
      }
   }
 }
}
//Main
Person becks = new Person("Becks",100);
Person posh = new Person("Posh",80);
GlobalFuncs.Marry(becks,posh);
becks.partner.setMoney(20);
posh.money;

class GlobalFuncs {
  static Person Marry(Person p1,Person p2) {
    p1.partner = p2;
    p2.partner = p1;
  }
}
```

Fig. 9. Translation of People program from Section 2.1

6 Related Work

BabyJ^T followed the motivation of Cecil[3], an object based language developed by Craig Chambers. BabyJ^T is aimed at a commercially successful language, JavaScript, rather than also define another language. Our work has been guided by the formal definitions and proof of soundness rather than aim to have a large number of features.

BeCecil[4] is a formalization of a variant of Cecil, however it does not have incremental typing and no soundness proof is given.

Strongtalk[8, 2] is an incremental typechecker for SmallTalk[10]. It uses structural rather than named types. It supports parameterized types and classes. However, the approach is class based rather than starting object based and using types to obtain a class based program.

7 Conclusions and Future Work

The style of programming we aim to support with this work frees the programmer of the burden of types in the early stages of development. Incremental typing can

reveal errors and inconsistencies not apparent in the early stages of development. Once fully typed, a program is guaranteed not to generate runtime type errors.

In further work we aim to study what guarantees can be given in the presence of $*$, eg. some functions contain occurrences of $*$, but are not used during execution.

We also plan to extend BabyJ to allow nesting of functions and function definitions as expressions. We want to extend the type system of BabyJ^T to allow subtyping and parametric types. A promising route would add delegation to BabyJ, and consider whether to represent this in the translation to class based as delegation as in [13, 15] or through subclasses.

Optimization of the translation from BabyJ^T to Java$_0$ in particular, when it can be shown that a functional member of an object is not re-assigned, and therefore the case statement and `int` field in the translated code can be removed.

We want to create tools to allow interactive type replacement and an implementation of the translation from BabyJ^T to Java$_0$.

8 Acknowledgements

We would also like to thank Rui Caldas, Robert Chatley, Neil Datta, Susan Eisenbach, John Knottenbelt, Mark Skipper, Matthew Smith, and colleagues at Imperial College Department of Computing, for many useful suggestions, and the ESOP'02 referees for constructive criticism.

References

1. M. Abadi and L. Cardelli. An imperative object calculus, 1995.
2. Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 215–230, 1993.
3. C. Chambers. The cecil language: Specification and rationale. Technical Report TR-93-03-05, 1993.
4. Craig Chambers and Gary T. Leavens. BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing. In *The Fourth International Workshop on Foundations of Object-Oriented Languages, FOOL 4, Paris, France*, 1996.
5. Christopher Anderson, Sophia Drossopoulou. Java$_S$ - A Subset of Java - report - `http://www.binarylord.com/work`.
6. Christopher Anderson, Sophia Drossopoulou. δ an imperative object based calculus. Presented at the workshop USE in 2002, Malaga, 2002.
7. William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135. ACM Press, 1990.
8. Dave Griswold et al. The strongtalk system report - `http://www.cs.ucsb.edu/projects/strongtalk/pages/index.html`.
9. David Flanagan. *JavaScript - The Definitive Guide.* O'Reilly, 1998.
10. Adele Goldberg and David Robson. *SmallTalk-80 The Language and its Implementation.* ADDWES, 1983.

11. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.
12. Javascript Language Reference. `http://developer.netscape.com/docs/manuals/js/client/jsguide/ClientGuideJS13.pdf`.
13. Günter Kniesel. *Darwin – Dynamic Object-Based Inheritance with Subtyping.* PhD thesis, CS Dept. III, University of Bonn, Germany, 2000.
14. Microsoft. C-sharp language specification `http://msdn.microsoft.com/vstudio/techinfo/articles/upgrade/Csharpdownload.asp`.
15. Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, Malaga, Spain, 2002.
16. Bjarne Stroustrup. *The design and evolution of C++.* ACM Press/Addison-Wesley Publishing Co., 1994.
17. Guido van Rossum. Python reference manual. Report, 2001.

A Definitions

A.1 Lookup Functions

We define relation $\mathsf{t} \approx \mathsf{t}'$, and auxiliary functions $\mathcal{R}$ and $\mathcal{D}$ used in the typing of expressions. The relation $\mathsf{t} \approx \mathsf{t}'$, used to determine if two types are compatible, holds if $\mathsf{t} = \mathsf{t}$', $\mathsf{t} = *$ or $\mathsf{t}' = *$.

For example, (`Person` $\approx$ $*$) and (`Person` $\approx$ `Person`) holds but, (`Person` $\approx$ `Frog`), does not. $\mathcal{R}(\mathbb{P},\mathsf{ts})$ returns the type descriptor of a constructor and is defined as:

$$\mathcal{R}(\mathbb{P},\mathsf{ts}) = \begin{cases} \mathsf{td} & \text{if } \mathbb{P}(\mathsf{ts}) = \texttt{constructor f(...)\{ this : td; ...\}} \\ * & \text{if } \mathsf{ts} = * \\ \mathit{Udf} & \text{otherwise} \end{cases}$$

For example,$\mathcal{R}(\mathbb{P}_1$,`Person`$) =$ `[name:string,...]` , $\mathcal{R}(\mathbb{P}_1,*) = *$ and $\mathcal{R}(\mathbb{P}_1$,`Marry`$)$ $= \mathcal{U}df$. $\mathcal{D}(\mathbb{P},\mathsf{ts},\mathsf{m})$ extracts the type of a member in a type descriptor and is defined as:

$$\mathcal{D}(\mathbb{P},\mathsf{ts},\mathsf{m}) = \begin{cases} \mathsf{t}_k & \text{if } \mathcal{R}(\mathbb{P},\mathsf{ts}) = [\mathsf{m}_1 : \mathsf{t}_1 ... \mathsf{m}_n : \mathsf{t}_n] \text{ and } \mathsf{m}_k = \mathsf{m}\ (1 \leq\ k\ \leq\ n) \\ * & \text{if } \mathcal{R}(\mathbb{P},\mathsf{ts}) = * \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

For example, $\mathcal{D}(\mathbb{P}_1$,`Person`,`name`$) =$`string` , $\mathcal{D}(\mathbb{P}_1$,`Person`,`shoe`$) = \mathcal{U}df$ and $\mathcal{D}(\mathbb{P}_1,*$,`name`$) = *$.

A.2 Heap update

We define heap update, $\mathsf{H}' = \mathsf{H}\{\iota.\mathsf{m} \lhd \mathsf{v}\}$, where H' is identical to H except that in ι member m is overridden/added by v:

$$\begin{array}{l} \mathsf{H}\{\iota.\mathsf{m} \lhd \mathsf{v}\}(\iota)(\mathsf{m}) = \mathsf{v}, \\ \mathsf{H}\{\iota.\mathsf{m} \lhd \mathsf{v}\}(\iota)(\mathsf{m}') = \mathsf{H}(\iota)(\mathsf{m}') \ \text{ if } \mathsf{m}' \neq \mathsf{m} \end{array}$$

A.3 Unique Functions

We now define unique functions for BabyJ and BabyJ^T $\vdash \mathrm{P}\diamond_u$ and $\vdash \mathbb{P}\diamond_u$ respectively:

$$\frac{\forall\, \mathsf{f} : \mathrm{P} \;\; = \mathrm{P}_1\ \texttt{function}\ \mathsf{f}(\mathsf{x})\{...\}\ \mathrm{P}_2 \qquad \mathrm{P} \;\; = \mathrm{P}_3\ \texttt{function}\ \mathsf{f}(\mathsf{x})\{...\}\ \mathrm{P}_4 \implies \mathrm{P}_1 = \mathrm{P}_3, \mathrm{P}_2 = \mathrm{P}_4}{\vdash \mathrm{P}\diamond_u}$$

$$\frac{\forall\, \mathsf{f} : \mathbb{P} \;\; = \mathbb{P}_1\ \_\ \mathsf{f}(...)\{...\}\ \mathbb{P}_2 \qquad \mathbb{P} \;\; = \mathbb{P}_3\ \_\ \mathsf{f}(...)\{...\}\ \mathbb{P}_4 \implies \mathbb{P}_1 = \mathbb{P}_3, \mathbb{P}_2 = \mathbb{P}_4}{\vdash \mathbb{P}\diamond_u}$$

A.4 *Filter*

The function $\mathcal{F}ilter$ is used to find the assignments to `this` in the body of a constructor:

$$\mathcal{F}ilter(\mathsf{e}) = \begin{cases} \{\mathsf{m}\} \cup \mathcal{F}ilter(\mathsf{e}''), & \text{if } \mathsf{e} \equiv \texttt{this.m} = \mathsf{e}'; \mathsf{e}'' \\ & \quad \text{and } \texttt{this} \text{ does not appear in } \mathsf{e}' \\ \emptyset & \text{otherwise} \end{cases}$$

The right hand side of the assignment cannot refer to `this`. The reason for this restriction is that when checking the body of a constructor we use an environment that gives `this` the type of the object the constructor is creating. If we did not have that restriction then it would be possible to select members from `this` before they have been created by the constructor body. Consider the following example:

```
constructor A {
 this:[m1:int,m2:int,m3:int]
 this.m1 = 2;
 this.m2 = this.m3;
 this.m3 = 10;
}
```

This would type check in an environment where `this` has type `A` but execution would get stuck because `this.m3` has not been created yet. Hence applying $\mathcal{F}ilter$ to the body of `A` would produce `{m1}` and ignore `this.m2 = this.m3;` as it uses `this` on the right hand side. Therefore, the `A` is not a valid constructor function with respect to ($\mathbb{P} \vdash \mathsf{f}\diamond_c$). Similarly $\mathcal{F}ilter$(`this.m1 = 2; foo(); this.m2 = 3`) would produce `{m1}` and ignore the rest of the assignments to `this` as they have been "broken" by the call to `foo()`. However the following is a valid constructor for `A`:

```
constructor A {
 this:[m1:int,m2:int,m3:int]
 this.m1 = 2;
 this.m2 = 3;
 this.m3 = 4;
 foo();
}
```

as $\mathcal{F}ilter($`this.m1 = 2;this.m2 = 3;this.m3 = 4;foo()`$) = \{$`m1,m2,m3`$\}$ and this conforms to `this:[m1:int,m2:int,m3:int]`.

B From BabyJ to BabyJ^T

We now give a translation function, $\mathcal{E}$, that given a BabyJ program returns a fully annotated BabyJ^T version of the program where all annotations are $*$.

$$\mathcal{E} :: (\mathit{Program} \rightarrow \mathit{TypedProgram}) \cup (\mathit{FuncDecl} \rightarrow \mathit{TypedFuncDecl}) \cup (\mathit{Body} \rightarrow \mathit{TypedBody})$$

$$\mathcal{E}(\mathrm{P}) = \mathcal{E}(\mathrm{D}_1)...\mathcal{E}(\mathrm{D}_n) \quad \text{where } \mathrm{P} = \mathrm{D}_1...\mathrm{D}_n$$

$$\mathcal{E}(\texttt{function f(x) \{B\}}) = \texttt{function} * \texttt{f(x : *) \{} \mathcal{E}(\mathrm{B}) \texttt{\}}$$

$$\mathcal{E}(\texttt{var y;B }) = \texttt{this} : *; \texttt{var y} : *; \mathrm{B}$$

It is clear that if we applied $\mathcal{E}$ to Figure 1 we would get Figure 4.

C Definitions for soundness of BabyJ^T type system

Figure 10 gives definitions for strongly typed programs, functions and environment. Figure 11 gives definitions for agreement between heaps, stacks and values.

We define the function $\mathcal{S}trip$ used to translate a BabyJ^T program into a corresponding BabyJ program. Types do not affect execution of BabyJ^T programs, therefore, we can remove them and use the operational semantics of BabyJ to execute BabyJ^T programs:

$$\mathcal{S}trip :: (\mathit{TypedProgram} \rightarrow \mathit{Program}) \cup (\mathit{TypedFuncDecl} \rightarrow \mathit{FuncDecl}) \cup (\mathit{TypedBody} \rightarrow \mathit{Body})$$

$$\mathcal{S}trip(\mathbb{P}) = \mathcal{S}trip(\mathbb{D}_1)...\mathcal{S}trip(\mathbb{D}_n) \quad \text{where } \mathbb{P} = \mathbb{D}_1...\mathbb{D}_n$$

$$\mathcal{S}trip(\texttt{\_ f(x : t) \{ } \mathbb{B} \texttt{ \}}) = \texttt{function f(x) \{ } \mathcal{S}trip(\mathbb{B}) \texttt{ \}}$$

$$\mathcal{S}trip(\texttt{this : ts; var y : t;} \mathrm{B}\ ) = \texttt{var y;} \mathrm{B}$$

$$\frac{\mathsf{t} = \mathsf{f} \neq *}{\vdash \mathsf{t} \diamond_S} \qquad \frac{\mathsf{t} = (\mathsf{ts}, \mathsf{t}', \mathsf{t}'') \quad \vdash \mathsf{ts} \diamond_S \quad \vdash \mathsf{t}' \diamond_S \quad \vdash \mathsf{t}'' \diamond_S}{\vdash \mathsf{t} \diamond_S}$$

$$\frac{\mathsf{f} = \texttt{function t f}(\mathtt{x} : \mathsf{t}_1) \quad \{\texttt{this} : \mathsf{ts}; \texttt{ var y} : \mathsf{t}_1'; \mathsf{e};\} \quad \vdash \mathsf{t} \diamond_S \quad \vdash \mathsf{ts} \diamond_S \quad \vdash \mathsf{t}_1 \diamond_S \quad \vdash \mathsf{t}_1' \diamond_S}{\vdash \mathsf{f} \diamond_{mS}}$$

$$\frac{\mathsf{f} = \texttt{global t f}(\mathtt{x} : \mathsf{t}_1) \quad \{\texttt{var y} : \mathsf{t}_1'; \mathsf{e};\} \quad \vdash \mathsf{t} \diamond_S \quad \vdash \mathsf{t}_1 \diamond_S \quad \vdash \mathsf{t}_1' \diamond_S}{\vdash \mathsf{f} \diamond_{gS}}$$

$$\frac{\begin{array}{l}\mathsf{f} = \texttt{constructor f}(\mathtt{x} : \mathsf{t}_1) \quad \{\texttt{this} : [\mathsf{m}_1 : \mathsf{t}_1' ... \mathsf{m}_p : \mathsf{t}_p']; \texttt{ var y} : \mathsf{t}_1''; \mathsf{e};\} \\ \forall\, \mathsf{t} \in \{\mathsf{t}_1' ... \mathsf{t}_p'\} : \vdash \mathsf{t} \diamond_S \\ \vdash \mathsf{t}_1 \diamond_S \quad \vdash \mathsf{t}_1'' \diamond_S\end{array}}{\vdash \mathsf{f} \diamond_{cS}}$$

$$\frac{\vdash \mathsf{f} \diamond_{mS}}{\vdash \mathsf{f} \diamond_S} \qquad \frac{\vdash \mathsf{f} \diamond_{cS}}{\vdash \mathsf{f} \diamond_S} \qquad \frac{\vdash \mathsf{f} \diamond_{gS}}{\vdash \mathsf{f} \diamond_S}$$

$$\frac{\begin{array}{l}\vdash \mathbb{P} \diamond \\ \forall \mathsf{f} \in \mathit{dom}(\mathbb{P}) : \vdash \mathsf{f} \diamond_S\end{array}}{\vdash \mathbb{P} \diamond_S}$$

$$\frac{\forall\, \mathtt{x} : \Gamma(\mathtt{x}) \neq \mathcal{U}\mathit{df} \quad \Longrightarrow \quad \Gamma(\mathtt{x}) \neq *}{\vdash \Gamma \diamond_S}$$

Fig. 10. Strong types, functions, programs and environments

$$\frac{\mathcal{R}(\mathbb{P}, \mathsf{t}) \neq \mathcal{U}\mathit{df}}{\mathbb{P}, \mathsf{H} \vdash \texttt{null} \lhd \mathsf{t}} \qquad \frac{\mathcal{R}(\mathbb{P}, \mathsf{f}) = \mathcal{U}\mathit{df}}{\mathbb{P}, \mathsf{H} \vdash \mathsf{f} \lhd \mathsf{t}} \qquad \frac{\begin{array}{l}\mathsf{H}(\iota) = [\![\mathsf{m}_1 : \mathsf{v}_1 ... \mathsf{m}_n : \mathsf{v}_n]\!] \\ \mathcal{R}(\mathbb{P}, \mathsf{t}) = [\mathsf{m}_1 : \mathsf{t}_1 ... \mathsf{m}_n : \mathsf{t}_n] \\ \mathbb{P}, \mathsf{H} \vdash \mathsf{v}_i \lhd \mathsf{t}_i \ \forall i \in \{1..n\}\end{array}}{\mathbb{P}, \mathsf{H} \vdash \iota \lhd \mathsf{t}}$$

$$\frac{\begin{array}{l}\mathsf{H}(\iota) = [\![...]\!] \quad \Longrightarrow \quad \mathbb{P}, \mathsf{H} \vdash \iota \lhd \mathcal{T}(\iota) \text{ (for all addresses)} \\ \Gamma(\texttt{this}) = \mathsf{t} \quad \Longrightarrow \quad \mathbb{P}, \mathsf{H} \vdash \mathsf{S}(\texttt{this}) \lhd \mathsf{t} \\ \Gamma(\mathtt{x}) = \mathsf{t} \quad \Longrightarrow \quad \mathbb{P}, \mathsf{H} \vdash \mathsf{S}(\mathtt{x}) \lhd \mathsf{t} \\ \Gamma(\mathtt{y}) = \mathsf{t} \quad \Longrightarrow \quad \mathbb{P}, \mathsf{H} \vdash \mathsf{S}(\mathtt{y}) \lhd \mathsf{t}\end{array}}{\mathbb{P}, \Gamma, \mathcal{T} \vdash \mathsf{H}, \mathsf{S} \diamond}$$

Fig. 11. Agreement between heaps, stacks and values

D Translation schema for ⟦ ⟧

We now give the definition of the translation function ⟦ ⟧. There is a case of ⟦ ⟧ defined for each element of the syntax of BabyJ^T. The program, $\mathbb{P}$, and a one-to-one mapping, $\mathcal{M}$, from function identifiers to integers, are used by the translation. The mapping $\mathcal{M}$ assigns each of the function identifiers in $\mathbb{P}$ a unique integer value to be used in the translated code to keep track of which translated member function to use.

D.1 Translation of programs

The translation of a BabyJ^T program $\mathbb{P}$, consists of the translation of all constructors and global functions. Each of the translated global functions is placed in the class `GlobalFuncs`. As they are static they can be accessed globally in the translated program (via a static method call `GlobalFuncs.m()`).

$$
\begin{array}{l}
\llbracket \mathbb{P} \rrbracket_{(\mathbb{P},\mathcal{M})} \triangleq \llbracket \mathsf{f}_1 \rrbracket_{(\mathbb{P},\mathcal{M})} \ \cdots \ \llbracket \mathsf{f}_n \rrbracket_{(\mathbb{P},\mathcal{M})} \\
\quad \texttt{class GlobalFuncs \{} \\
\quad\quad \llbracket \mathsf{f}'_1 \rrbracket_{(\mathbb{P},\mathcal{M})} \ \cdots \ \llbracket \mathsf{f}'_m \rrbracket_{(\mathbb{P},\mathcal{M})} \\
\quad \texttt{\}} \\
\quad\quad \text{where } \{\mathsf{f}_1 ... \mathsf{f}_n\} = \{\mathsf{f} \mid \mathbb{P}(\mathsf{f}) = \texttt{constructor...}\ \} \\
\quad\quad\quad\quad \{\mathsf{f}'_1 ... \mathsf{f}'_m\} = \{\mathsf{f} \mid \mathbb{P}(\mathsf{f}) = \texttt{global...}\ \}
\end{array}
$$

D.2 Translation of constructors

A constructor is translated into a Java$_0$ class definition containing the translation of the type descriptor and a constructor. The constructor contains the translation of the BabyJ^T constructor body.

$$
\begin{array}{l}
\llbracket \texttt{constructor } \mathsf{f}(\mathsf{x} : \mathsf{t})\ \{\ \texttt{this} : \mathsf{td}; \texttt{var } \mathsf{y} : \mathsf{t}'; \mathsf{e}_{\mathsf{body}} \} \rrbracket_{(\mathbb{P},\mathcal{M})} \triangleq \\
\quad \texttt{class } \mathsf{f}\ \texttt{\{} \\
\quad\quad \llbracket \mathsf{td} \rrbracket_{(\mathbb{P},\mathcal{M})} \\
\\
\quad\quad \mathsf{f}(\llbracket \mathsf{t} \rrbracket\ \mathsf{x})\ \texttt{\{} \\
\quad\quad\quad \llbracket \mathsf{t}' \rrbracket\ \mathsf{y}; \\
\quad\quad\quad \llbracket \mathsf{e}_{\mathsf{body}} \rrbracket_{(\mathbb{P},\mathcal{M})} \\
\quad\quad \texttt{\}} \\
\quad \texttt{\}}
\end{array}
$$

D.3 Translation of member functions

Member functions are represented as branches of a Java$_0$ case statement. The case statement is contained within a method with the same signature as the member function in the translated constructor.

$$
\begin{array}{l}
(\!|\texttt{function t f(x : t}_1\texttt{)\{ this : ts; var y : t}'_1\texttt{; e}_{\mathsf{body}}\texttt{\}}|\!)_{(\mathbb{P},\mathcal{M})} \triangleq \\
\quad \texttt{case } \mathcal{M}(\mathsf{f}) : \{ \\
\qquad (\!|\texttt{t}'_1|\!)\ \texttt{y}; (\!|\texttt{e}_{\mathsf{body}}|\!)_{(\mathbb{P},\mathcal{M})} \\
\quad \}
\end{array}
$$

D.4 Translation of global functions

A global function is translated into a static method whose return and formal parameter types correspond to the translation of those defined in the function definition. The method will be placed inside the `GlobalFuncs` class.

$$
\begin{array}{l}
(\!|\texttt{global t f(x : t}_1\texttt{)\{ var y : t}'_1\texttt{; e}_{\mathsf{body}}\texttt{\}}|\!)_{(\mathbb{P},\mathcal{M})} \triangleq \\
\qquad (\!|\texttt{t}|\!)\ \texttt{f}((\!|\texttt{t}_1|\!)\ \texttt{x})\ \{ \\
\qquad\quad (\!|\texttt{t}'_1|\!)\ \texttt{y}; (\!|\texttt{e}_{\mathsf{body}}|\!)_{(\mathbb{P},\mathcal{M})} \\
\qquad \}
\end{array}
$$

D.5 Translation of Expressions

The translation of expressions is straightforward, with BabyJ method call translated to Java$_0$ method call due to the handling of member assignment. Of interest is the global call which translates to a static call of the corresponding method in the class `GlobalFuncs`. Also, function identifiers are translated into the value mapped by $\mathcal{M}$.

$$
\begin{array}{ll}
(\!|\texttt{this}|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq \texttt{this} \\
(\!|\texttt{x}|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq \texttt{x} \\
(\!|\texttt{y}|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq \texttt{y} \\
(\!|\texttt{f}|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq \mathcal{M}(\mathsf{f}) \\
(\!|\texttt{e}_1\texttt{; e}_2|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq (\!|\texttt{e}_1|\!)_{(\mathbb{P},\mathcal{M})}\texttt{; }(\!|\texttt{e}_2|\!)_{(\mathbb{P},\mathcal{M})} \\
(\!|\texttt{f(e)}|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq \texttt{GlobalFuncs.f}((\!|\texttt{e}|\!)_{(\mathbb{P},\mathcal{M})}) \\
(\!|\texttt{e}_1\texttt{.m = e}_2|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq (\!|\texttt{e}_1|\!)_{(\mathbb{P},\mathcal{M})}\texttt{.m } = (\!|\texttt{e}_2|\!)_{(\mathbb{P},\mathcal{M})} \\
(\!|\texttt{x = e}|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq \texttt{x} = (\!|\texttt{e}|\!)_{(\mathbb{P},\mathcal{M})} \\
(\!|\texttt{new f(e)}|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq \texttt{new f}((\!|\texttt{e}|\!)_{(\mathbb{P},\mathcal{M})}) \\
(\!|\texttt{e}_1\texttt{.m(e}_2\texttt{)}|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq (\!|\texttt{e}_1|\!)_{(\mathbb{P},\mathcal{M})}\texttt{.m}((\!|\texttt{e}_2|\!)_{(\mathbb{P},\mathcal{M})})
\end{array}
$$

D.6 Translation of type descriptors

Each member of the type descriptor (including function types) for a given constructor corresponds to a field of the translated class. Member function types in the type descriptor are also translated into a method definition that contains a case statement which distinguishes between the various possible cases for the function - these are known because the candidates must have the same type as the function member. The field associated with a function member is of type `int` and keeps track of the latest assignment. By using a field with the same identifier

as the associated method we avoid having to use type information when translating assignments to members. Therefore, assignment to a function member is represented as assignment to the `int` field and function call is represented as method call.

$$
\begin{array}{ll}
(\!|\mathsf{td}|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq (\!|\mathsf{m}_1 : \mathsf{t}_1|\!)_{(\mathbb{P},\mathcal{M})} ... (\!|\mathsf{m}_n : \mathsf{t}_n|\!)_{(\mathbb{P},\mathcal{M})} \\
& \quad \text{where } \mathsf{td} = [\mathsf{m}_1 : \mathsf{t}_1 ... \mathsf{m}_n : \mathsf{t}_n] \\
(\!|\mathsf{m} : \mathsf{ts}|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq (\!|\mathsf{ts}|\!)\ \mathsf{m};
\end{array}
$$

$$
\begin{array}{ll}
(\!|\mathsf{m} : (\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3)|\!)_{(\mathbb{P},\mathcal{M})} & \triangleq \texttt{int}\ \mathsf{m}; \\
& \quad (\!|\mathsf{t}_3|\!)\ \mathsf{m}((\!|\mathsf{t}_2|\!)\ \mathtt{x})\ \{ \\
& \qquad \texttt{switch}\ (\mathsf{m})\{ \\
& \qquad\quad (\!|\mathsf{f}_1|\!)_{(\mathbb{P},\mathcal{M})}\ \cdots\ (\!|\mathsf{f}_m|\!)_{(\mathbb{P},\mathcal{M})} \\
& \qquad \} \\
& \quad \} \\
& \quad \text{where } \{\mathsf{f}_1 ... \mathsf{f}_m\} = \{\mathsf{f}\ |\mathbb{P}(\mathsf{f}) = \texttt{function}\ \mathsf{t}_3\ \mathsf{f}(\mathtt{x} : \mathsf{t}_2)\{\texttt{this} : \mathsf{t}_1 ...\}\}
\end{array}
$$

D.7 Translation of types and environments

The translation of types is the identity function up to function types which are mapped to the Java$_0$ `int` type.

$$
(\!|\mathsf{t}|\!) \triangleq \begin{cases} \texttt{int}, \text{ if } \mathsf{t} = (\_, \_, \_) \\ \mathsf{t}, \text{otherwise} \end{cases}
$$

The environment $(\!|\Gamma|\!)$ maps elements in the domain of Γ to the translation of the corresponding element in the range of Γ.

E Operational Semantics of BabyJ

Figures 12 and 13 give the operational semantics for generation and propagation of exceptions.

F Proving Soundness of Translation (| |)

F.1 Preservation of static semantics

In order to be prove Theorem 2 we must extend the theorem to all subterms of $\mathbb{P}$. This can then be proved by induction on the typing rules. The case for expressions is the most interesting and is stated below.

The translation preserves types up to function types. That is, if a BabyJ^T expression has type t with respect to $\mathbb{P}$ and environment Γ, and e is translated into a Java$_0$ expression e' that has type t' with respect to $(\!|\mathbb{P}|\!)$ and $(\!|\Gamma|\!)$ then t =t' if t is not a function type and t'=`int` if t is a function type.

$$\frac{\mathsf{e}, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{null}, \mathsf{H}', \mathsf{S}'}{\begin{array}{l}\mathsf{e.m}, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{nullPntrExc}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e.m} = \mathsf{e}', \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{nullPntrExc}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e.m}(\mathsf{e}'), \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{nullPntrExc}, \mathsf{H}', \mathsf{S}'\end{array}}$$

$$\frac{\mathsf{S}(\mathtt{x}) = \mathcal{U}df}{\begin{array}{l}\mathtt{x}, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{stuckErr}, \mathsf{H}, \mathsf{S} \\ \mathtt{x} = \mathsf{e}, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{stuckErr}, \mathsf{H}, \mathsf{S}\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}' \\ \mathsf{v} \neq \ \mathtt{null} \\ \mathsf{v} \notin \text{Addr or } \mathsf{H}(\mathsf{v}) = \mathcal{U}df\end{array}}{\begin{array}{l}\mathsf{e.m}, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{stuckErr}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e.m} = \mathsf{e}', \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{stuckErr}, \mathsf{H}', \mathsf{S}'\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \iota, \mathsf{H}', \mathsf{S}' \\ \mathsf{H}'(\iota)(\mathsf{m}) = \mathcal{U}df\end{array}}{\mathsf{e.m}, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{stuckErr}, \mathsf{H}', \mathsf{S}'}$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \iota, \mathsf{H}_1, \mathsf{S}_1 \\ \mathsf{e}_2, \mathsf{H}_1, \mathsf{S}_1 \ \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}' \\ \mathsf{H}'(\iota)(\mathsf{m}) = \mathcal{U}df\end{array}}{\mathsf{e}_1.\mathsf{m} = \mathsf{e}_2, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{stuckErr}, \mathsf{H}', \mathsf{S}'}$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathsf{v}, \mathsf{H}', \mathsf{S}' \\ \mathsf{v} \neq \ \mathtt{null} \\ \mathsf{v} \notin \text{Addr or } \mathsf{H}(\mathsf{v}) = \mathcal{U}df \text{ or } \mathsf{H}(\mathsf{v})(\mathsf{m}) = \mathcal{U}df\end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{stuckErr}, \mathsf{H}', \mathsf{S}'}$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \iota, \mathsf{H}_1, \mathsf{S}_1 \\ \mathsf{e}_2, \mathsf{H}_1, \mathsf{S}_1 \ \rightsquigarrow \mathsf{v}', \mathsf{H}', \mathsf{S}' \\ \mathsf{H}'(\iota)(\mathsf{m}) = \mathsf{f} \\ \mathrm{P}(\mathsf{f}) = \mathcal{U}df\end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{stuckErr}, \mathsf{H}', \mathsf{S}'}$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathsf{v}', \mathsf{H}', \mathsf{S}' \\ \mathrm{P}(\mathsf{f}) = \mathcal{U}df\end{array}}{\mathsf{f}(\mathsf{e}), \mathsf{H}, \mathsf{S} \ \rightsquigarrow \mathtt{stuckErr}, \mathsf{H}', \mathsf{S}'}$$

Fig. 12. Operational Semantics - generation of exceptions

$$\frac{\mathsf{e},\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}'}{\begin{array}{l}\mathtt{x} = \mathsf{e},\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}' \\ \mathsf{f}(\mathsf{e}),\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}' \\ \mathsf{e}.\mathsf{m},\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}' \\ \mathsf{e}.\mathsf{m} = \mathsf{e}',\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}' \\ \mathsf{e}.\mathsf{m}(\mathsf{e}'),\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}'\end{array}}$$

$$\frac{\mathsf{e}_1,\mathsf{H},\mathsf{S} \rightsquigarrow \iota,\mathsf{H}_1,\mathsf{S}_1 \quad \mathsf{e}_2,\mathsf{H}_1,\mathsf{S}_1 \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}'}{\begin{array}{l}\mathsf{e}_1.\mathsf{m} = \mathsf{e}_2,\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}' \\ \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2),\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}'\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e}_1,\mathsf{H},\mathsf{S} \rightsquigarrow \iota,\mathsf{H}_1,\mathsf{S}_1 \\ \mathsf{e}_2,\mathsf{H}_1,\mathsf{S}_1 \rightsquigarrow \mathsf{v}',\mathsf{H}_2,\mathsf{S}' \\ \mathsf{H}_2(\iota)(\mathsf{m}) = \mathsf{f} \\ \mathrm{P}(\mathsf{f}) = \mathtt{function}\ \mathsf{f}(\mathtt{x})\{\mathtt{var}\ \mathtt{y};\ \mathsf{e}'\} \\ \mathsf{e}',\mathsf{H}_2,\{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}', \mathtt{y} \mapsto \mathcal{U}df,\} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}''\end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2),\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}'}$$

$$\frac{\begin{array}{l}\mathsf{e},\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{v}',\mathsf{H}_1,\mathsf{S}' \\ \mathrm{P}(\mathsf{f}) = \mathtt{function}\ \mathsf{f}(\mathtt{x})\{\mathtt{var}\ \mathtt{y};\ \mathsf{e}'\} \\ \mathsf{e}',\mathsf{H}_1,\{\mathtt{this} \mapsto \mathcal{U}df, \mathtt{x} \mapsto \mathsf{v}', \mathtt{y} \mapsto \mathcal{U}df\} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}''\end{array}}{\mathsf{f}(\mathsf{e}),\mathsf{H},\mathsf{S} \rightsquigarrow \mathsf{dv},\mathsf{H}',\mathsf{S}'}$$

Fig. 13. Operational Semantics - propagation of exceptions

Lemma 1. *For any BabyJ^T expression* e*, strongly typed program* $\mathbb{P}$ *and environment* Γ*, if*

- $\mathbb{P}, \Gamma \vdash \mathsf{e} : \mathsf{t}$*, and*
- $(\!|\mathsf{e}|\!)_{(\mathbb{P},\mathcal{M})} = \mathsf{e}$*', and*

then

- $(\!|\mathbb{P}|\!)_{(\mathbb{P},\mathcal{M})}, (\!|\Gamma|\!) \vdash \mathsf{e}' : (\!|\mathsf{t}|\!)$

Proof is by induction on the application of the typing rules.

F.2 Preservation of dynamic semantics

The following definitions are used in proving the preservation of dynamic semantics.

Definition 1. *Let b be a bijection between finite sets of address A,A' and* $\mathcal{M}$ *a one-to-one mapping from function identifiers to integers. We define agreement of values,* $\mathcal{M} \vdash \mathsf{v} \approx_b \mathsf{v}'$ *if*

- $\mathsf{v} = \iota$ *for some* ι *and* $\mathsf{v}' = b(\iota)$ *or,*

– $\mathsf{v} = \mathsf{f}$ *and* v*'=i,* $\mathcal{M}(\mathsf{f}) = i$ *or,*
– $\mathsf{v} = \mathsf{v}$*'*

Definition 2. *Let* $\mathbb{P}, \Gamma, \mathcal{T} \vdash \mathsf{H}, \mathsf{S} \diamond$ *and* $(\mathbb{P})_{(\mathbb{P},\mathcal{M})}, (\Gamma) \vdash \mathsf{R}, \mathsf{T} \diamond$*, we say* R*,* T *are the translation of* H*,* S *with respect to* b *and write* $\mathbb{P}, \Gamma, \mathcal{M} \vdash \mathsf{H}, \mathsf{S} \approx_b \mathsf{R}, \mathsf{T}$ *if*

– b *is a bijection between* $\{\iota | \mathsf{H}(\iota) \neq \mathcal{U}df\}$ *and* $\{\iota | \mathsf{R}(\iota) \neq \mathcal{U}df\}$
– $\mathcal{M}$ *is a one-to-one mapping from function identifiers to integers.*
– *for all* ι*, if* $\mathsf{H}(\iota) = [\![\mathsf{m}_1 : \mathsf{v}_1 ... \mathsf{m}_n : \mathsf{v}_n]\!]$ *then*
  $\mathsf{R}(b(\iota)) = [\![\mathsf{m}_1 : \mathsf{v}'_1 ... \mathsf{m}_n : \mathsf{v}'_n]\!]^c$ *and* $\mathcal{M} \vdash \mathsf{v}_i \approx_b \mathsf{v}'_i \ \forall \ i \ \in \ \{1..n\}$
– $\mathcal{M} \vdash \mathsf{S}(\mathtt{x}) \approx_b \mathsf{T}(\mathtt{x})$ *,* $\mathcal{M} \vdash \mathsf{S}(\mathtt{y}) \approx_b \mathsf{T}(\mathtt{y})$*,* $\mathcal{M} \vdash \mathsf{S}(\mathtt{this}) \approx_b \mathsf{T}(\mathtt{this})$*,*

G Higher Order Examples

We now give a more interesting example that demonstrates the translation of BabyJ^T higher order functions, and passing functions as parameters and re-assigning members functions:

```
constructor A {
  this:[m1:(A,(A,int,int),(A,int,int)),m2:(A,int,int)]
  this.m1 = m3;
  this.m2 = m4;
  this;
}

function (A,int,int) m3(x:(A,int,int)) {
   this:A;
   this.m2 = x;
}

function int m4(x:int) {
  this:A;
  x;
}

function int m5(x:int) {
  this:A;
  x*x;
}

//Main
Var a:A;
a = new A();
a.m2(10); // 3
a.m1(m5);
a.m2(11); // 6
```

And now the corresponding translation into Java_0 with mapping $\mathcal{M}$ defined as: $\mathcal{M}(\texttt{m3}) = 1$,$\mathcal{M}(\texttt{m4}) = 2$, $\mathcal{M}(\texttt{m5}) = 3$.

```
class A {
    int m1;

    int m1(int x) {
        switch(m1) {
            case 1: {
                this.m2 = x
            }
        }
    }

    int m2;

    int m2(int x) {
        switch(m2) {
           case 2: {
                x;
           }
           case 3: {
                x*x
           }
        }
    }

    A() {
      this.m1 = 1;
      this.m2 = 2;
    }
}

//Main
A a = new A();
a.m2(10); // 3
a.m1(3); //changes member m2
a.m2(11); // now returns 6
```